basic

1900 digital computer concepts

AN I·C·T PROGRAMMED TEXT

basic digital computer concepts

International Computers and Tabulators Limited

Programmed Learning Department
Bradenham Manor, nr High Wycombe, Bucks

First Edition . . . . . April 1965

Printed in Great Britain by

The Trigon Press, Great Missenden, Buckinghamshire.

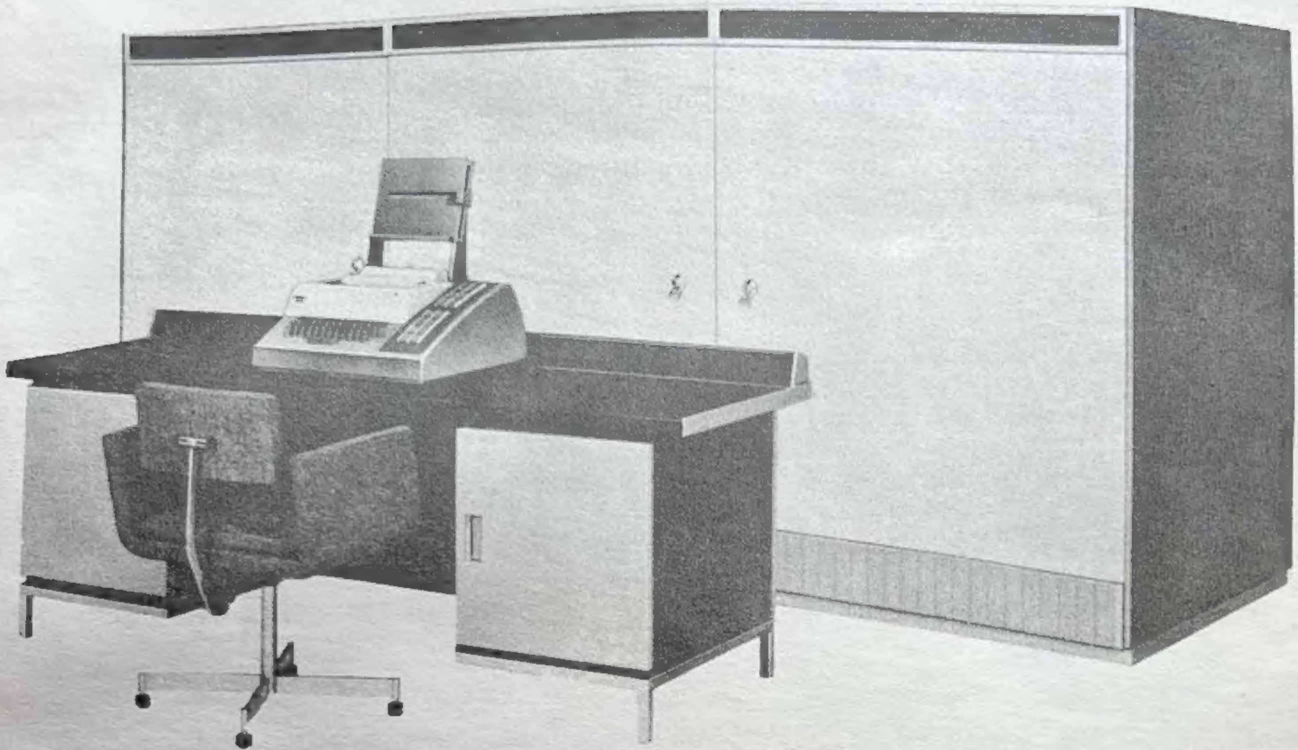
TO THE READER

This book sets out to give you a high-speed introduction to digital computers and to show you that this highly technical subject makes surprisingly simple overall sense.

Because we could assume only a lively intelligence and keen interest on the part of you, the reader, we start from scratch by introducing the idea of computers in business as if to a young person who is just beginning his career. Thereafter, the ICT Series 1900 provides most of the examples and these first steps are seen mainly in terms of that specific computer.

Although the material is presented for quick, easy reading rather than hard study, you will want to hold on to the main points. To help you do so, the end of each section is a checkpoint at which you're invited to test yourself on that section's ten or so main points. In addition to this, occasionally an individual point presents an opportunity for you to write an answer.

When you've finished the course satisfactorily, you'll be ready—should you need to do so—to learn how to program a specific computer.



The I.C.T. 1904/1905/1906/1907 Central Processor

CONTENTS

Section I contains nothing about computer hardware. Instead, it introduces the environment in which modern digital computers mostly operate.

If you have some familiarity with business practice, or if you have gathered from general sources some idea of how computers perform business data processing tasks, you may want to skip Section I.

In Section II, you are still 'external', as it were, to the computer proper. There you consider a variety of input and output media with particular reference to their speed of working.

Thus, you will see what sort of work a digital computer does and at what sort of pace.

In Sections III and IV, you will see how the computer does its processing and how the programmer commands its performance.

SECTION I

COMPUTERS IN BUSINESS

There are two quite distinct families of computers, analogue computers and digital computers.

Analogue computers are very different from digital computers.

Digital work is ordinary alphabetical and number work with the usual arithmetic processes.

You create an **analogue** situation by giving values some actual, as opposed to numerical, form. When asked how long was the one that got away, the fisherman may reply digitally, 'Eighteen inches', or he may give an analogue answer with his hands. A motor-car's milometer is digital but the speedometer is analogue. Despite the numbers on the dial and our translation into numerical terms, speeds are represented by the position of a pointer and not by numerical digits.

When you use a slide-rule, you manipulate actual lengths of wood (or whatever). Thus, a slide-rule is a sort of analogue computer.

Real analogue computers usually represent values by electrical potentials. They are used to simulate complicated industrial processes.

This course is about **digital** computers. Analogue computers belong to a different field.

Digital computers, then, work with numbers in much the same way as you do when you set out to solve a mathematical problem with paper and pencil.

They hold and manipulate numbers and alphabetical characters and they can be instructed to make logical decisions.

Modern digital computers tend to be all-purpose machines. Although individual machines **are** built, of course, for special purposes, for the general commercial market the trend is towards powerful and advanced computers of standard designs which may be ordered in a wide variety of sizes and configurations.

Thus, for a business application, the XYZ Computer might be ordered with a modest-sized central computing unit, with no less than eight magnetic-tape input/output units and with other units capable of holding hundreds of thousands of business records. For a **scientific** application, the XYZ Computer might still be the best available but for this purpose, it might be ordered with a **big** central computing unit, **minimal** input/output facilities, and **no** extra data-holding units.

Insert the appropriate letter, **a**, **b**, etc., from the key below into each box in this table:-

	Central computing unit	Input/output facilities	Extra data-holding units
Business application			
Scientific application			

a none, because permanent data-holding unnecessary

b fast and extensive, because very many business transactions to process

c big, because complicated and extensive calculations

d modest size, because calculations relatively simple

e necessary to hold hundreds and thousands of data records

f minimal, because only one set of input/output to each extensive calculation

Answer:

	Central computing unit	Input/output facilities	Extra data-holding units
Business application	d	b	e
Scientific application	c	f	a

If you were wrong at all, perhaps you need to read the last page a little more carefully.

These, of course, are only sample configurations but they serve to show some of the ways in which business and scientific computer requirements might differ.

Somewhat surprisingly, perhaps, business applications make greater demands on computers than do scientific applications. Even the more lengthy and complicated scientific calculations require no more than the basic arithmetic functions of adding, subtracting, multiplying and dividing plus the logical facilities of the computer. The less elaborate business calculation will require no less and business also demands a very fast and efficient throughput of records in vast numbers.

Business applications are the bread and butter of the data processing industry. Because of this, and because they usually exceed the requirement of science, we'll consider computers in business.

Let's begin by looking at the way in which computers get into the business world—how they're bought or rented by the user.

Business involves many routine tasks of the clerical type. When the volume of this sort of work begins to employ ever-increasing armies of clerical workers, each with a similar repetitive task, management may begin to consider more efficient methods.

Computers are not normally bought and sold 'over the counter'. Often, the prospective user invites an investigation by the computer manufacturer's experts who usually form an investigating team which includes managers and organisation experts from the prospective user's own staff. This joint team makes a series of studies of the business's organisation and procedures.

An early part of the investigations is called 'The Feasibility Study'. Clearly, it is important to establish, as soon as possible, whether or not the business would benefit from the installation of a computer.

If the computer installation is feasible, specific areas of the business are selected for computer application and detailed System Studies plan the computer systems to the point of actual machine specification.

At this stage, the prospective user is presented with a report of the proposed computer systems, complete with a detailed estimate of computer hardware and all other requirements.

If the report is accepted, a contract is signed and the user either buys or rents the computer.

The computer manufacturer also supplies a continuing service of support to the user's installation.

Delivery of the computer may be a year or eighteen months after the contract is signed; and this time will be needed for the selection and training of the user's computer installation staff, the writing of the actual computer programs, the preparation of forms and data records, and the preparation for the run-down of the old systems and take-over by the new computer systems.

Until the user's own staff are sufficiently established as a computer team, all this is initiated by the computer manufacturer's experts. In fact, the experts may need to take a leading part until after the computer systems are fully operational. And even thereafter, they usually assume a large degree of reponsibility for the success of the installation.

This is reassuring for a would-be user, but it means that a big computer selling company must maintain a large organisation of highly trained experts.

These include systems analysts—experts with considerable experience of business systems and computers, and computer programmers—experts on specific computers who must also know something of business systems.

As a new computer installation is planned, and as it develops, so the user's own team of systems analysts and computer programmers begins to form.

These two functions, systems analysis and computer programming, exemplify the two poles of the business computer world, the user's business on the one hand, and the machine—the actual computer hardware—on the other:

Systems work must look to the most efficient and profitable execution and control of the user's business.

The systems analyst and the programmer must each know quite a lot about the other's job.

Programming work is, of course, very much machine oriented. It entails writing the complete series of instructions which must be fed to a computer to enable it to perform a specific data processing task.

So, as well as employing engineers, computer operators and managers, both the computer manufacturer and the user need to select and train systems analysts and programmers.

What sort of previous experience would be most useful to new entrants to the computer field who hope to do systems or programming work? Please write down your answer before turning the page.

Answer: Business experience.

Without business experience, it's unusual to approach systems work from scratch. Certainly a systems adviser would feel ill-qualified without considerable experience. But, of course, this experience can be gained within the computer field.

Business experience is also a great help to the trainee programmer, and without it, he must pick it up as he goes along. This is what often happens, and when he's gained both programming and business experience, the programmer may become a systems man.

Fortunately, business is a rational activity and much can be learned taking a common-sense look at the broad picture. The newcomer who does this sometimes sees more than the business expert.

Since, on the part of you, the reader, we can assume no computer knowledge and no business knowledge, how can we discuss data-processing?

In fact, it's surprisingly easy. We've said that business practices make ordinary good sense. Let's regard data-processing as a job to be done and ignore the question as to whether the job is to be done manually or by computer.

Whatever sort of data you are considering, processing is based on main files. In computer terminology these are called MASTER FILES. In an ordinary office they may be called just 'files'.

If you were setting-up in business on your own, you might need to keep a list of your customers. You'd have a list of the lines of stock in which you dealt. Soon you might have a list of employees.

These are clearly the beginnings of three of your main files. They contain the main structure of your business, the more or less permanent data on which your business is based.

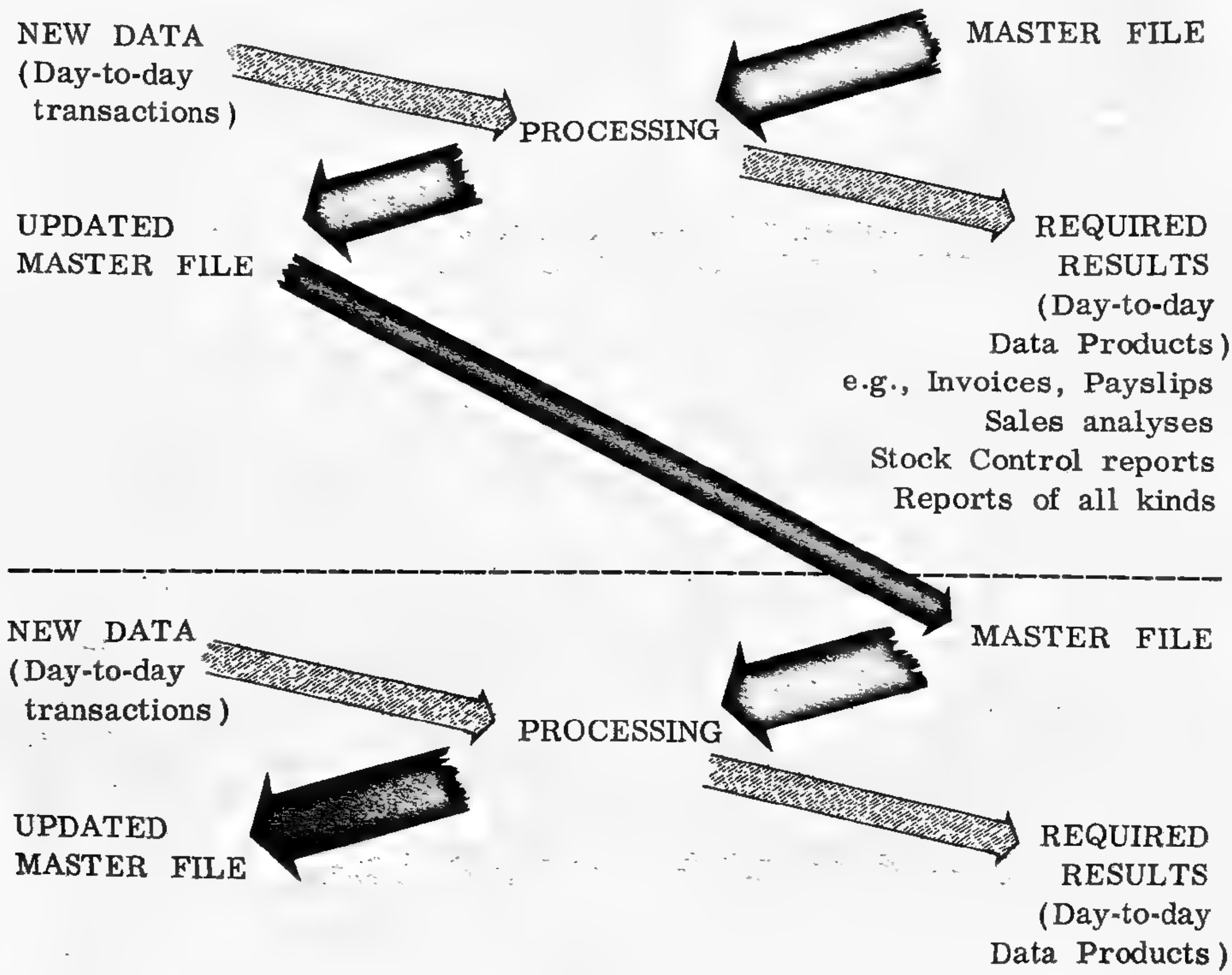
They do not contain the day-to-day transactions, but you use main file information constantly in dealing with day-to-day transactions.

Master or main file data will change, of course. Your stock file will record, for each stock line, the balance in kind (i.e. the quantity you hold at the moment). Each day-to-day transaction will alter that balance.

So, each day-to-day transaction relates to one or more master files. The master file provides information necessary in processing the transaction; and the transaction may alter data in the master file.

This business of a master file being altered by detailed transactions is called UPDATING the master file.

Master files being updated on the one hand, and day-to-day data being processed to a required end on the other—these provide data processing with an overall pattern:



. . . . and so on, everlastingly . . . .

You can see that the heavy black arrows trace the permanent structure of constantly updated master files. The actual workaday flow of business—always changing but always the same—the never-ending traffic of new transactions—this is the other axis.

Such a concept—general as it is—will help you when you're trying to understand actual business systems.

You'll have an initial plan of campaign—to sort out the permanent files from transaction documents.

In a manual system, the two sorts of documents may **look** alike. Permanent files may be paper records held in box-files, and similarly, the day-to-day transactions may also be gathered together and temporarily held in box-files.

So, even the physical 'file' can be a source of confusion.

In computer terminology, **all** data which goes into the computer is said to be in a file—in a punched card file, paper-tape file, or magnetic-tape file —whether it's a MASTER file or a collection of day-to-day transactions.

It's very necessary, therefore, to hold on to this idea of **MASTER** files and to distinguish between their more or less permanent data and the transaction data.

Everybody uses the word ‘records’ in a general sense to mean recorded information. In data-processing, a RECORD is a unit of data—the sense unit, if you like, rather as a sentence is the sense unit of language.

Here’s an example from a Stock File:

Part No.	Description	Unit	Balance Quantity	Balance Value		Date	Average Unit Price	Re-order Level	Re-order Qty	Last Re-order	Re-order Unit Price	Previous A.U.P.	
				£	s		pence				pence		
43592	Flange Sprocket	Gross	06847	04584	13	28 02 65	16 0·7	06000	6000	6 12 64	00159·1	00162·3	

(We’re not concerned, at the moment, with whether this is a manual record or a record in some computer medium, like a punched card or magnetic-tape.)

Is this a master file record or is it a transaction record?

Please write down ‘master file’ or ‘transaction’ before continuing.

It's a master file record—a permanent record which will be updated by transactions.

What about this record?

Part No.	Description	Date	Quantity Issued	Requisition No.
43592	Flange Sprocket	04 03 65	00050	B 7384

This is a transaction record. It records that on 4th March 1965, the storekeeper issued 50 gross of Flange Sprockets against Requisition No. B 7384. (Incidentally, this requisition number identifies the production department shop and job for which these stores were drawn.)

And what about this one?

Part No.	Description	Date	Quantity Recieved	Unit Price	Order No
43592	Flange Sprocket	26 03 65	06000	0163·7	730

This is a transaction record, too—the opposite sort of transaction to the last. This records that the storekeeper **received** a re-order consignment.

By the way, in computer terminology, the different items of information in a record are called FIELDS, e.g., the Part No. FIELD.

If we take from the master file one item's stock balance master record as at the end of last month—remember:

Part No.	Description	Unit	Balance Qty	Balance Value		Dat
				£	s	
43592	Flange Sprocket	Gross	06847	04584	13	

and gather together this month's stock issue records for the same item:

Part No.	Description	Date	Quantity Issued	Rec
43592	Flange Sprocket	04 ⁞ 03 ⁞ 65	00050	

and this month's stock receipt records for the same item:

Part No.	Description	Date	Quantity Received	Uni
43592	Flange Sprocket	26 ⁞ 03 ⁞ 65	06000	0

We can __________ the master file record.

What's the word?

Write it down before continuing.

Update's the word. No doubt you got it.

Trouble is, the stores hold an enormous number of parts, and the storekeeper makes thousands of issue—and **receives** quite a lot of stuff—every month.

There aren't just a few stock balances to be updated, there are thousands of them; and thousands of issue and receipt records to sort out.

If the job had to be done manually, how would **you** tackle it?

Well, with all respect, the balances at the month-end would probably be a week or two out of date however you tackled it. Still, let's have a go.

In what sort of order would you expect to find the records in the master file? And in what sort of order would you arrange the issue and receipt records?

The master records and the transaction records—to match up with each other—must be in the same order of stock items.

Part Number order is perhaps most likely for the master file, and you'd sort the issue and receipt records to the same.

If both the master-file and the transaction records were in alphabetical order of part **Description**, this would do just as well.

This processing of sequentially arranged master records, matched with transaction records sorted to the same sequence, is called SEQUENTIAL PROCESSING (predictably enough!).

Now for the job itself:

For each stock item for which there were issues and receipts, **make a brief list of the processes** you'd carry out **to update the balance quantity only**.

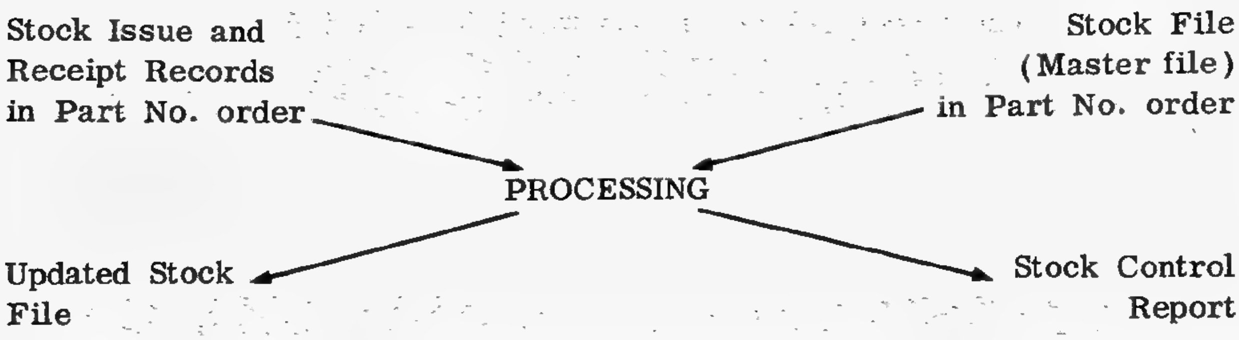
For each stock item, you'd

1. Get the old Balance Quantity.
2. Add the Receipt Quantities, if any.
3. Subtract the Issue Quantities, if any.

and this would leave you with the new Balance Quantity.

Perhaps, too, you'd want a Stock Control Report—a **document** showing the new updated master records—to save the managing director the fag of thumbing through the master file.

We can show your system like this:



As you can see, this conforms with the overall pattern of data processing which you saw earlier.

Consider the processing again.

This time, LET'S SUPPOSE YOU'VE DECIDED TO DEAL WITH **ONE TRANSACTION RECORD AT A TIME.**

START

1. Take the next Receipt or Issue Record
2. Get the Master Record for that stock item
3. If it's a Receipt Record, **Add** Quantity to old Balance Quantity
4. If it's an Issue Record, **Subtract** Quantity from old Balance Quantity

If there are no more transaction records,

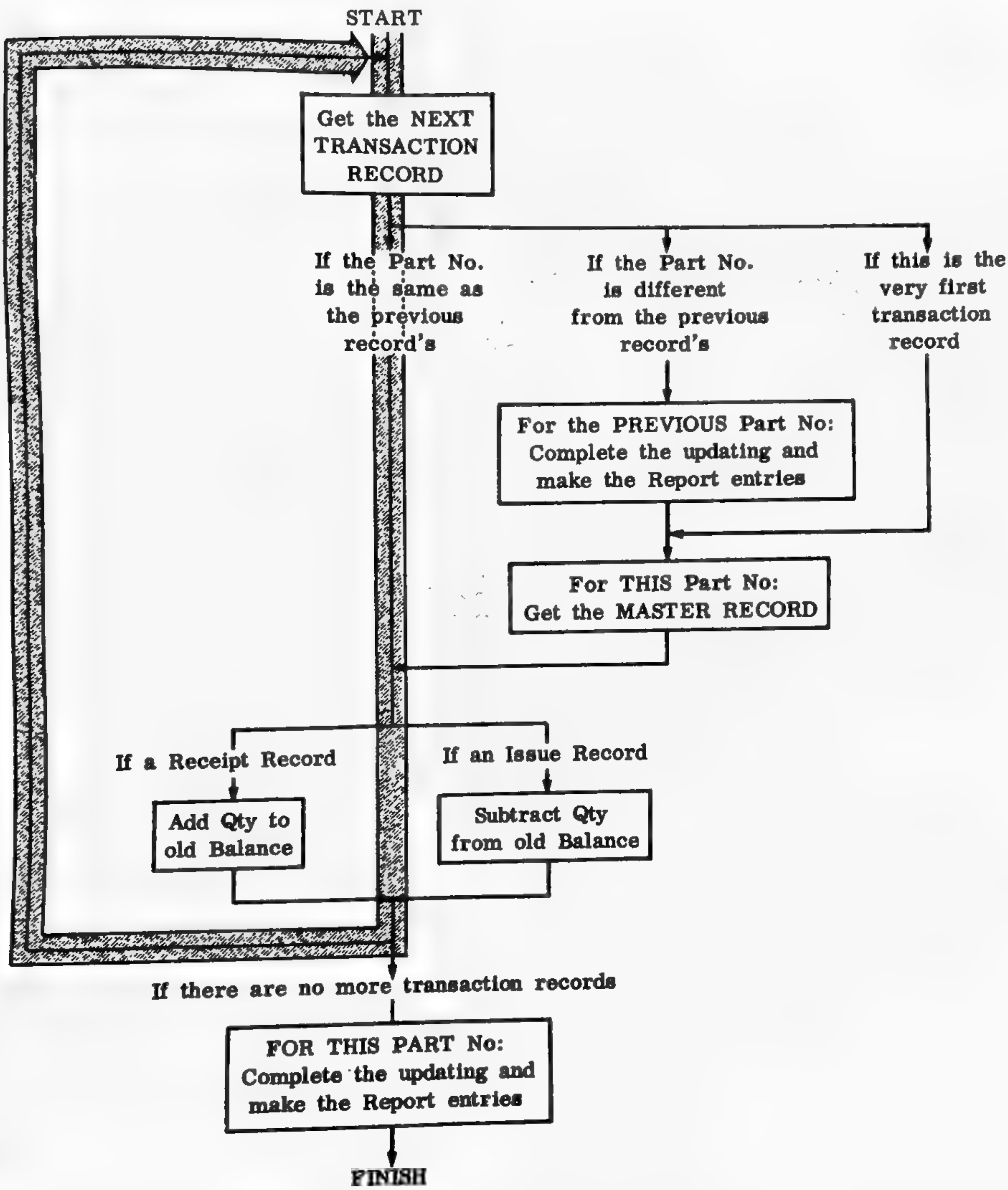
STOP

This diagram, or FLOW-CHART as it's called, could be improved.

It implies a search for the Master Record every time, even though there'll often be many transactions to the one master record. And it doesn't show what happens when you've finally got the new balance.

You can probably incorporate these points yourself.

Here's a more complete version; and now we're using boxes and showing alternative routes:



Notice how this flow-chart allows for the first record of all, the last record of all, and the end and beginning of each Part No. group.

You can regard the series of processes in the flow-chart as a list of instructions which tell you what to do.

A computer **PROGRAM** is precisely this—a list of instructions which the computer must carry out in the sequence indicated.

In computer terminology, the flow-chart on the opposite page takes the form of a **LOOP**. The **MAIN LOOP** is shown with the shaded band.

You can see that each transaction record, taking the appropriate route, is subjected to a single performance of the program. This is called a **PASS**.

The terms you've just met—PROGRAM, MAIN LOOP, PASS—are fairly obvious. The important points are these:

* For a computer, the list of instructions—the PROGRAM—must be absolutely complete. It must start properly, keep running, and end as planned, and it must make sure that the right instructions are carried out in all the eventualities it's supposed to cover.
The computer does nothing except by obeying a program instruction.

* The entire job of processing data records with one program is called a **RUN**. This entails loading the **program** of instructions into the computer and then feeding all the appropriate data records to the computer.

* A digital computer on commercial-type work will, in a typical run, process thousands of records, each of which is subjected to a **PASS** through the program.

You've now met:

I The SYSTEMS concept.

II The PROGRAM concept.

On page 22, you saw part of a business system. On page 24, you've just seen a very simple program.

Systems flow-charts, which may describe manual systems or computer systems, show the series of processing jobs which must take place on files and records.

A computer **systems** flow-chart will therefore consist, largely, of a series of computer runs.

A **program** flow-chart shows how one of those runs will be carried out by the computer. It's a planning aid which helps the programmer to write the computer's foolproof list of instructions.

To be simple, in this first section we've made our examples commonplace and unambitious—in fact, we've committed a gross 'undersell' on computers.

Don't imagine a systems analyst merely traces a manual system and sticks in a computer to do the same processing jobs each as a computer run.

Don't imagine a computer to be only a sort of super, fantastically high-speed clerk.

A computer can do many jobs at once and some jobs which were never before considered because they were clearly beyond the capacity of whole armies of clerks.

Management bases its decisions on business data, and computers can provide the sort of up-to-the-minute analyses and estimates which are otherwise quite inaccessible. This service can, and does, revolutionise business.

CHECKPOINT

1. 'This book is about analogue computers.' Is this statement true or false?

2. 'Computers for business applications are less elaborate than computers for scientific applications.' Is this statement true or false?

3. What do we call the study which determines whether or not a computer should be used?

4. Who investigates and re-plans systems—on the business side?

5. Who writes the detailed list of computer instructions?

6. What do we call the main files of a business when they are held in a computer medium?

7. What happens to these files in the course of computer processing?

8. What do we call a single performance of a computer program?

9. What do we call the computer's performance of an entire data-processing job?

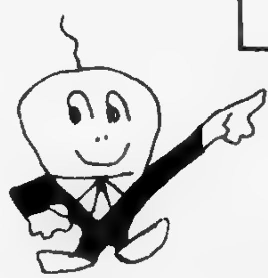
10. What sort of chart is drawn to plan a computer job?

CHECKPOINT ANSWERS

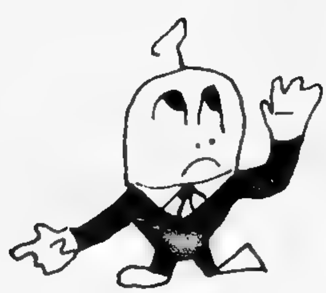
1. False.
2. False.
3. The Feasibility Study.
4. A Systems Analyst.
5. A Computer Programmer.
6. Master files.
7. They are **updated**.
8. A **pass**.
9. A **run**.
10. A program flow-chart.

You are ready to go on to the next section

If you scored eight or more,



If you scored less than eight,



You need to look at this section again

SECTION II

PERIPHERALS

In this Section, you take a look at the devices which put data into the computer and those which record the results.

We are particularly interested in the speed at which these devices work.

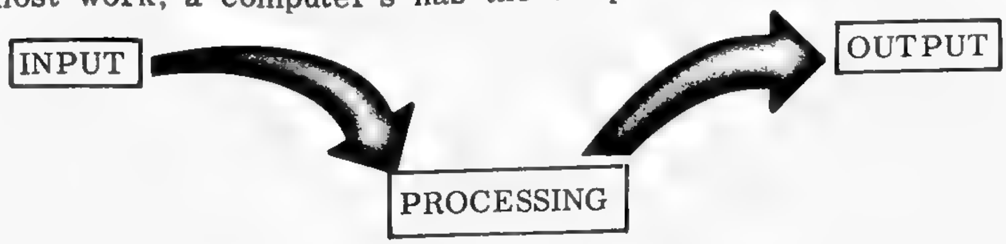
You must remember that the processing of data—the calculations, etc—takes place between input and output. Computers process very quickly indeed—in **millionths of a second**—and usually have to waste time waiting for the next input data.

This is one reason why we're so interested in the **speed** of input and output devices—to make the most of the computer's tremendous processing speed.

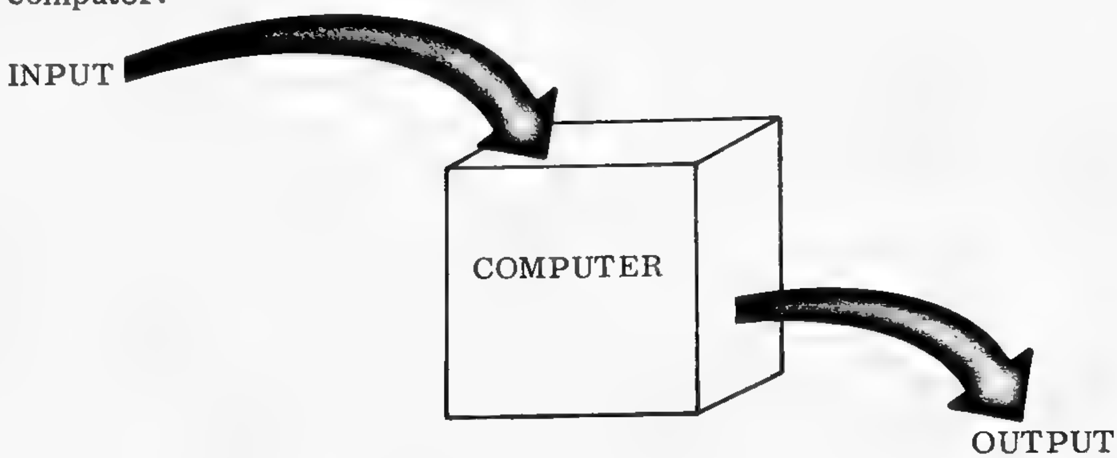
The other reason is all to the same purpose of efficient working, but instead of looking to the computer's fantastic ability to process data faster than it can be fed in and out, we look to the thousands upon thousands of records we'd like the computer to process for us. Naturally, we want the throughput rate of these records to be as high as possible so that the business gains maximum benefit from up-to-date information.

If the computer actually **processes** data so very quickly, then clearly the speed of our input and output media will often be a critical factor in determining our overall rate of working.

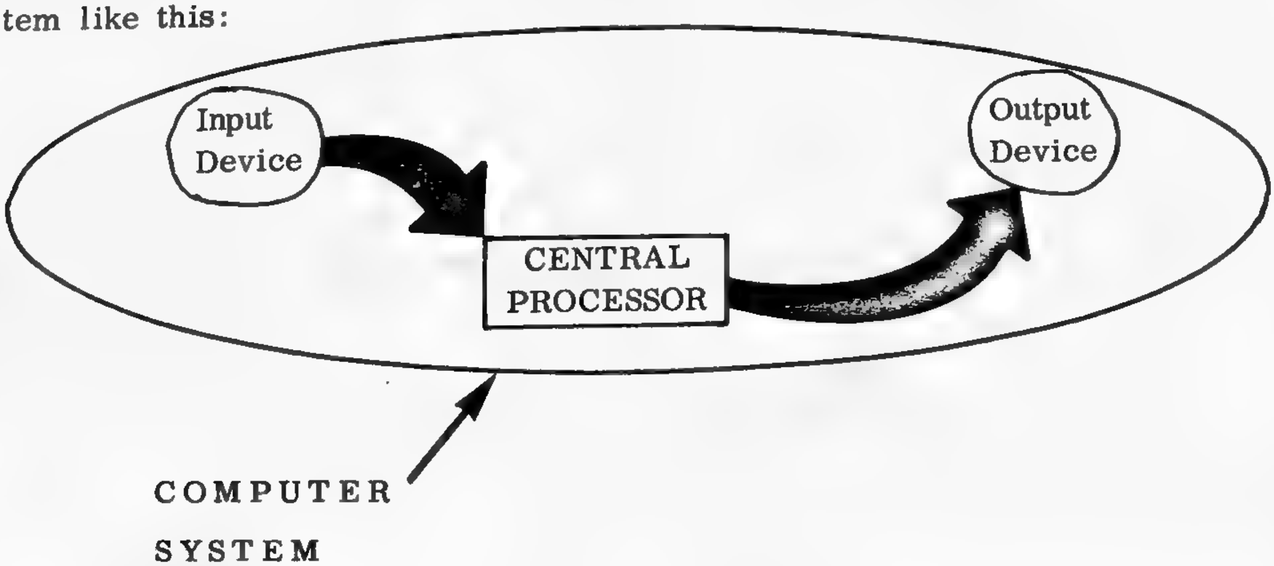
Like most work, a computer's has the simple overall pattern:-



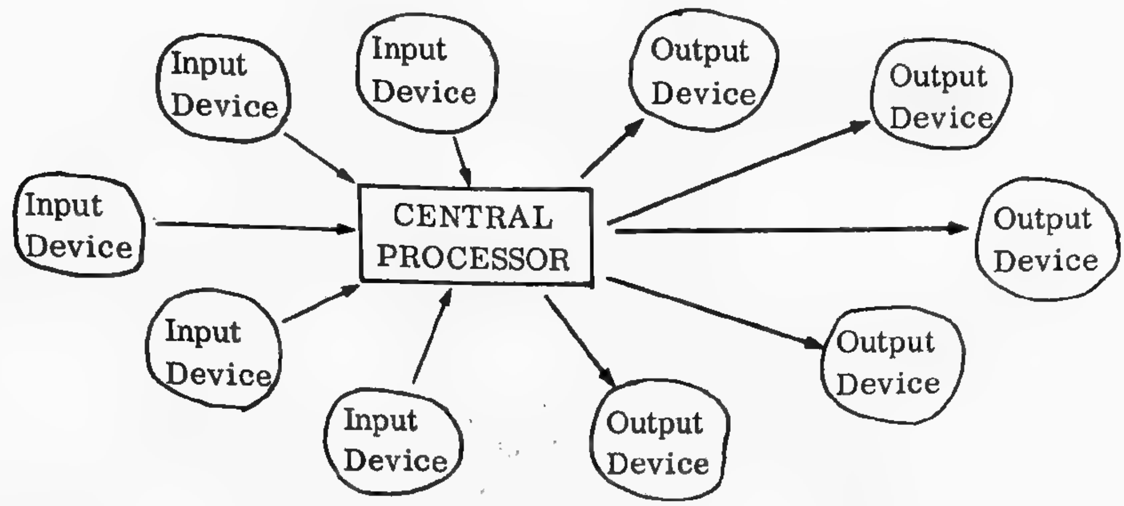
You might suppose, quite reasonably, that only the processing part is the actual computer.



But in fact, we regard input and output devices as part of the computer system like this:



Computer systems have a variety of input and output devices:



Because they're dotted **around** the central processor, we call the input and output devices **peripheral units** or just **peripherals**.

Now, computers process data and input devices are the various means of feeding in the data factors. Output devices are the various means of presenting the results.

Before going on to look at some examples of input and output devices, i.e., peripherals, from the ICT Series 1900 Computer, we must make a couple of overall points.

Input and output media belong to two categories:

1. Slow-speed media.

2. High-speed media.

There are two other divisions you can make:

a) Those input media which can be created manually, and ouput media which can be manually read.

b) Those media which can be created only by computer and can be read only by computer.

Fortunately, this still makes only two categories, not four. The divisions coincide.

Slow speed media are also those which can be both created and read manually.

High speed media can be created or read only by computer.

Now for the actual devices.

THE I.C.T. 1911 CARD READER

This is an input peripheral—a means of getting data into the central processor.

On demand of the program, it reads data punched in standard 80-column cards at speeds of up to 960 cards a minute.

This means that the Card Reader responds to a **card read** instruction in the program. When such an instruction is reached a card is read and its data is transferred into the central processor.

Conversion from the card code for numbers and letters to the pattern in which they are held by the central processor is quite automatic.

At 960 cards a minute, the ICT 1911 can read 76,800 numerical digits or letters in sixty seconds.

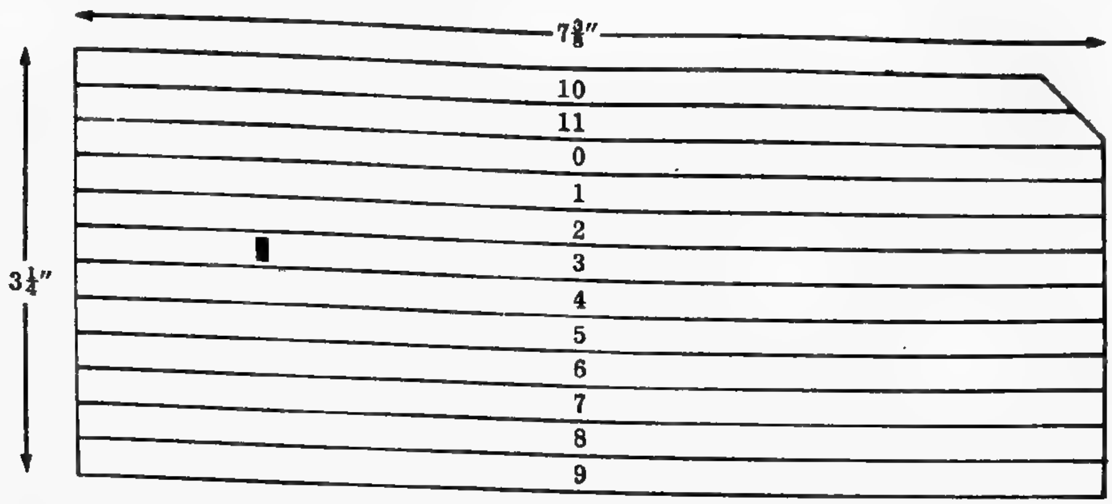
Allowing for unused card columns, this could mean approximately 1,000 characters a second. But whilst this is enormously fast **card** reading, magnetic-tape reading can give us up to 96,000 characters a second!

These two figures—1,000 characters a second for very fast card reading and 96,000 characters a second for very fast magnetic-tape reading—help us to keep input/output speeds in perspective.

We regard card reading as a slow speed input media.

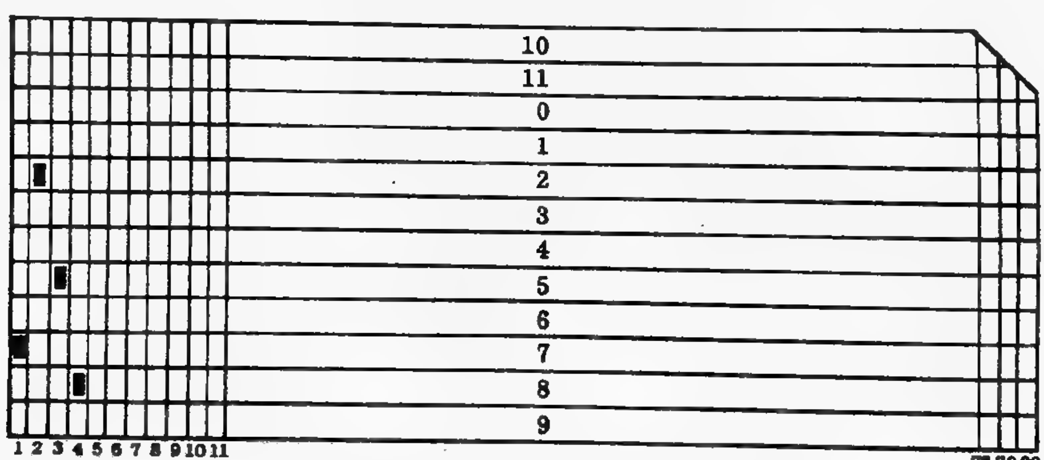
Punched cards provide a simple and straightforward introduction to computer records. Because of this and because they're long established and still going strong, they deserve your more detailed attention.

Standard 80-column punched cards are approximately $7\frac{3}{8}$ inches long by $3\frac{1}{4}$ inches wide. A card has twelve rows or levels, each with its numerical significance.



The diagram shows a hole punched in the card. You can see what number the hole represents.

Because it's divided into 80 columns, a card can hold up to 80 characters.

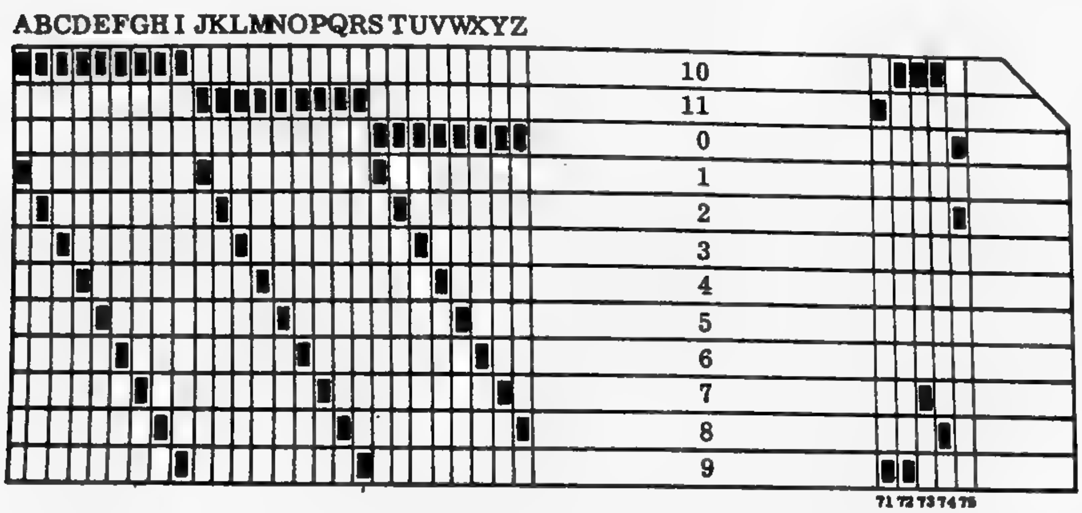


Write down the four-figure number punched in the first four of the 80 columns, before continuing on the next page.

You should have written 7258. If you've not got that, have a look at page 37 again.

Alphabetical characters are represented by a simple extension of the numeric code. **Two** holes are punched in a single column for each alphabetical character.

Thus, the first nine letters of the alphabet are represented by the numerals 1 to 9, but in addition, each has the 10 hole punched. The numerals 1 to 9 are used again for the second nine letters of the alphabet, but this time each has the 11 hole punched ..... and so on, like this:



What word is punched in card columns 71 to 75?

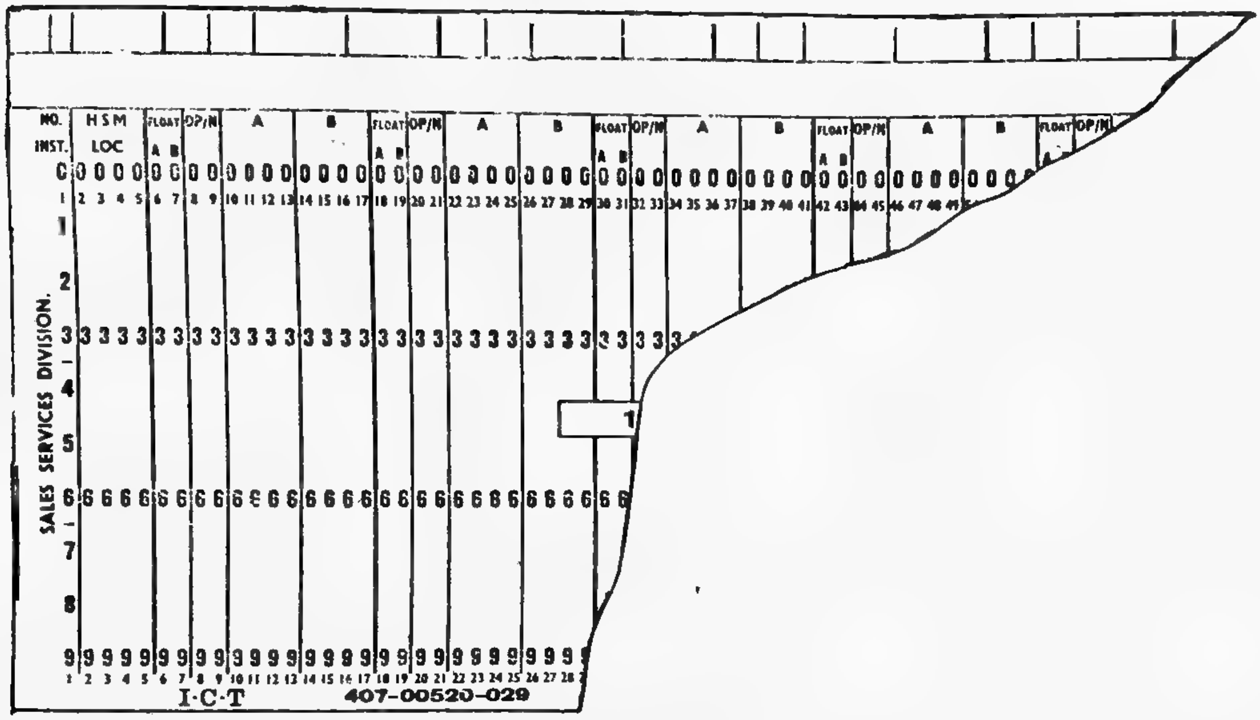
Write down your answer.

RIGHT. And if you were, well done!
If not, go back and try again.

In the latest computer systems, punched card readers sense the holes photo-electrically. A punched hole allows a beam of light to activate a photo-electric cell whereas solid card blocks the beam of light.

Photo-cell sensing allows very high card-reading speeds.

A group of card columns which holds an item of data is called a **field**, and punched cards are usually printed with their fields ruled off and labelled:

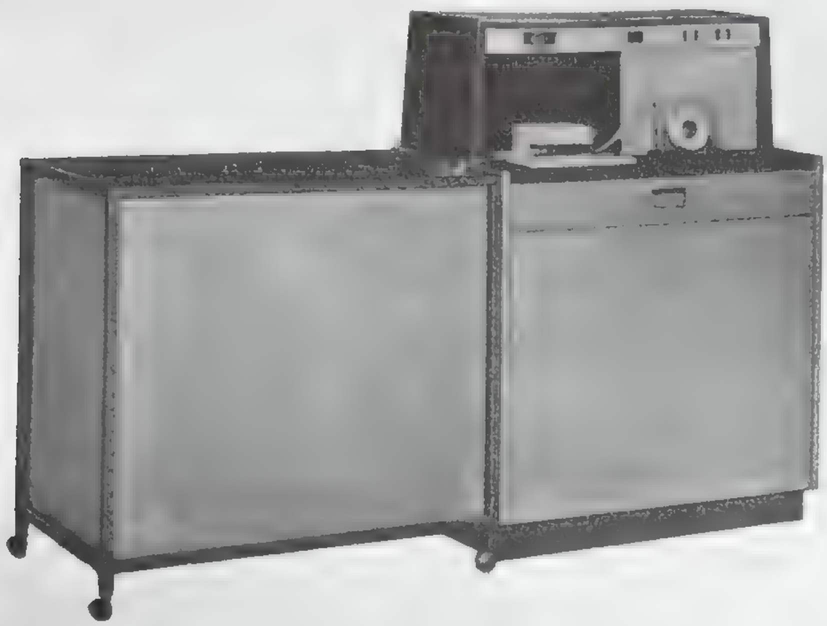


Like other forms of computer input and output, punched cards are designed to be read from, or punched to, by machine.

But punched cards can be created on a **key-board** machine, from the original sources of the data, whereas magnetic-tape records must originate from a computer.

Moreover, a punched card is a physically separate record. It can be easily got at—by hand if necessary, and, as you'd imagine, this can be useful.

THE I.C.T. 1916 PAPER-TAPE READER



This is another input peripheral—another means of getting data into the central processor.

On demand of the program, it reads data punched in 5, 6, 7 or 8-track paper-tape at speeds of up to 1000 characters a second—roughly equivalent to fast card reading.

The Paper-tape Reader responds to a **paper-tape read** instruction in the program. The instruction may specify the number of characters to be read from the paper-tape, or, alternatively, it may specify that characters should be read until an End-of-Block character is sensed.

This means that the paper-tape may be punched with a continuous series of characters or with groups of characters separated by an End-of-Block character. No **gaps** are needed between blocks of data on paper-tape.

1″ wide paper-tape costs 3/- per 1020-ft reel! It is easier to transport and much less bulky than punched cards. It is easily and cheaply produced as a by-product of commercial routines on desk adding-machines and from cash registers.

At 10 character frames to the inch, a reel of paper-tape can hold 104,000 characters—equivalent to the holding capacity of some 1,300 punched cards. But, whereas **any number** of punched cards can be fed to the computer non-stop, paper-tape reading involves reel changes. and reels may need re-winding. You can't get at an individual record with the same ease as with punched cards. Even so, cheapness, lack of bulk and the general convenience of handling and storing paper-tape make it an attractive proposition.

Unlike **magnetic**-tape, punched paper-tape records can be originated on keyboard machines in much the same way as punched cards and at much the same speed, cost and accuracy.

You can see that the user who is entering the computer field from scratch might find it very difficult to decide between punched cards and punched paper-tape as his 'prime' means of recording the data for computer input.

THE I.C.T. 1974 MAGNETIC-TAPE SYSTEM

This is, perhaps, the principal **high-speed** peripheral. It is at once a means of getting data into a central processor—an input peripheral, **and** a means of recording results—an output peripheral.

You can, therefore, program to **read** from magnetic tape, or **write** to it.

Data characters are recorded in frames across magnetic-tape on much the same coding principle as on paper-tape. Instead of punched holes, of course, the code is signalled by magnetised spots.

The I.C.T. 1974 magnetic-tape system will read or write 96,000 characters a second.

As you saw in the picture, the ICT 1974 Magnetic-tape System houses a single tape-deck in a separate free-standing unit.

To handle the normal run of magnetic-tape work, a computer needs at least four such single tape-decks.

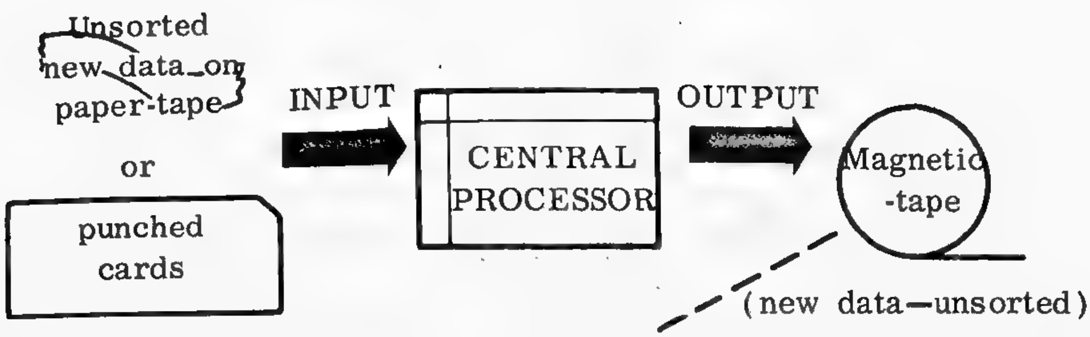
Because magnetic-tape work is so fast, a common task—illustrated on the next page—is to

1. WRITE TO MAGNETIC-TAPE new, **unsorted** data from cards or paper-tape.

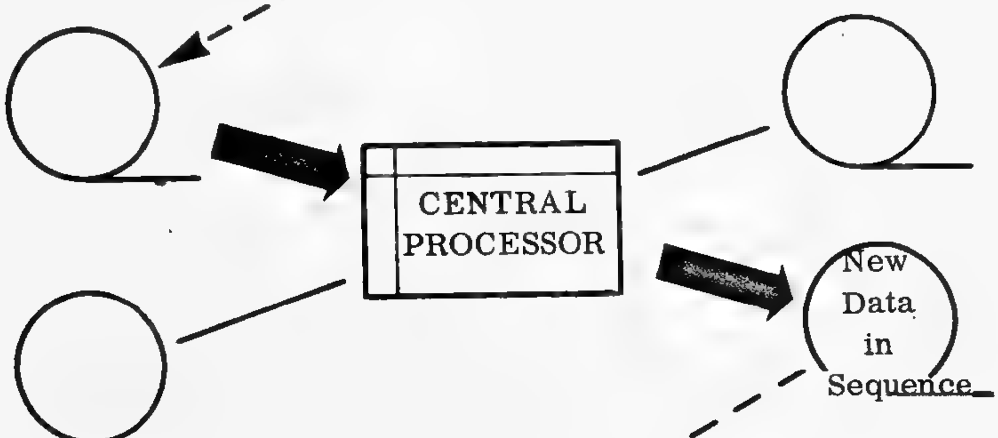
2. MAGNETIC-TAPE SORT the data to the master-file sequence at very high speed, thus saving, in the case of cards, many hours of slow, cumbersome sorting on punched-card sorting machines.

3. UPDATE the magnetic-tape master-file with the new data now sorted on magnetic-tape.

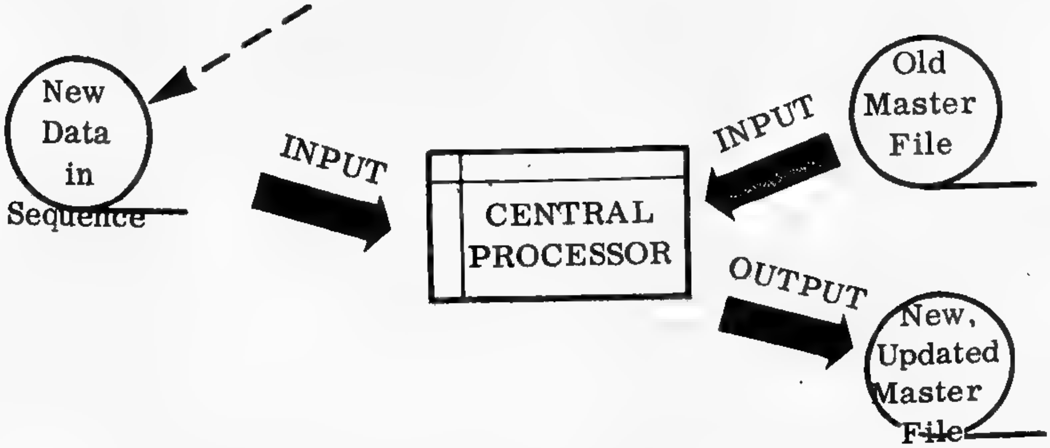
1.



2. The new data must now be sorted—an operation which normally requires the use of four magnetic-tape decks.



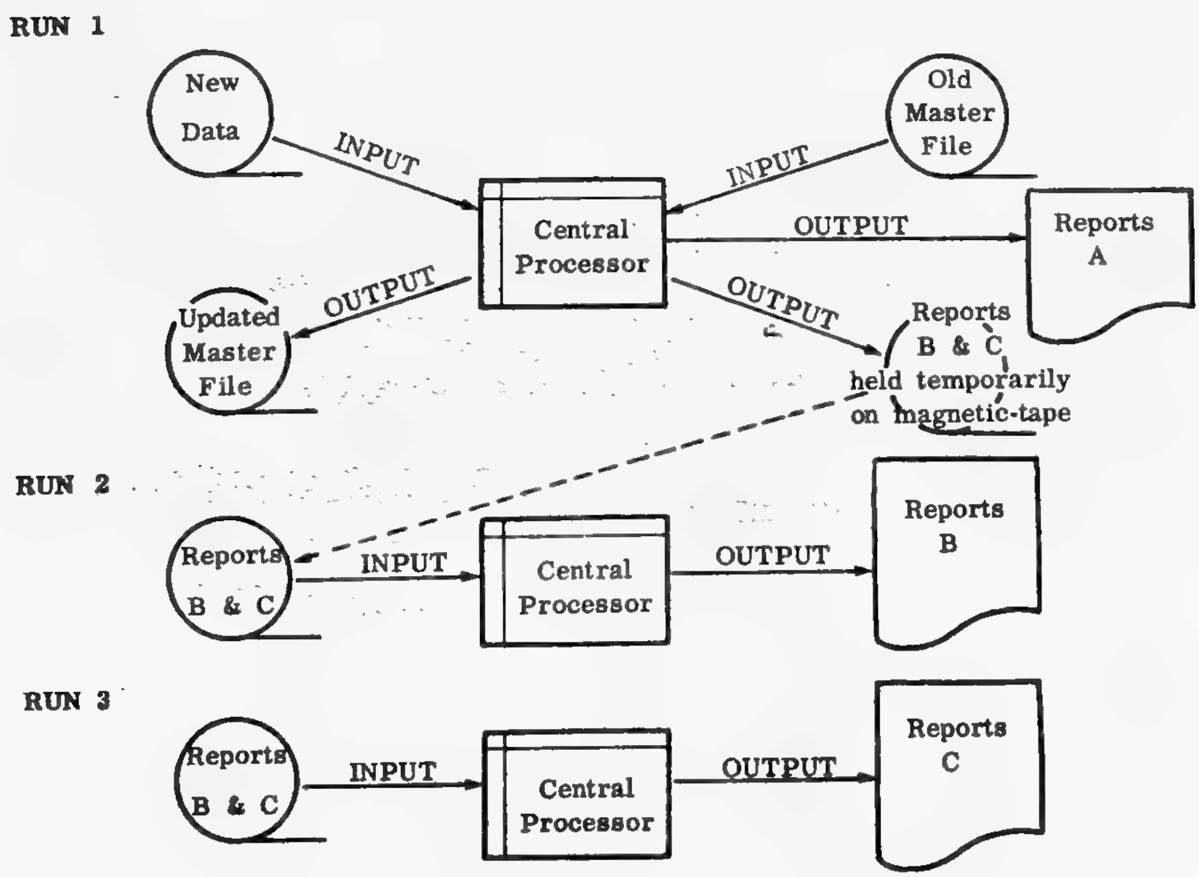
3. Now in sequence, the new data may add to, subtract from, or generally amend an existing file of records. These permanent files are called **master files**.



So far, it may seem that the programmer is concerned to get data onto reels of magnetic-tape and use it to create other reels of magnetic-tape— which is a bit like a dog chasing its own tail.

Clearly, at least some of a computer's output must finally be in a form which can be read by **people**. (This will remain true, of course, only until the world is **completely** automated!)

Thus, even while creating an updated master file tape a computer may be producing printed reports on another peripheral, a printer. (Printed analyses of all kinds, payslips, share certificates, stock control reports, etc. etc. etc.)



THE I.C.T. 1933 PRINTER

This is an output peripheral. A computer's printer is usually called a **line** printer because a printing instruction in the program normally causes a horizontal **line** of characters to be printed on the continuous stationery.

The continuous stationery is 'spaced'—i.e., the paper moves up one line—before a line is printed.

A line of print comprises 96, 120 or 160 printing positions according to machine specifications.

On each of these printing positions across the paper, any character can be printed from a range of 64,—numerals, alphabet and special symbols.

The ICT 1933 Printer will print 1100 lines a minute if the full range of 64 characters is required. But, if some of the special characters are not needed and the range of characters can be limited to 48, then the printing speed is 1350 lines a minute.

Work out a character rate per second which might be achieved for printed output and write down your answer before you continue.

Well, all sorts of variations are possible, but you're really trying to see what a fast rate might be.

Let's say you have 120 printing positions of which you use 100 for characters—the rest being used to give horizontal spacing.

At top speed, this is 135,000 characters a minute—**a rate per second of 2,250 characters**.

Now, 1350 lines a minute is extremely fast printing, but writing to magnetic-tape is still forty times faster! Remember too, that the central processor works at **microsecond** speeds whereas, even at 1350 lines a minute, to print a line takes over 40 **milliseconds** (**thousandths** of a second) i.e. over 40,000 microseconds.

You can see that **printing** output data direct from the central processor is slow compared with fast peripheral transfers to or from magnetic-tape.

This sort of printing—direct from the central processor—is called **on-line**. Any peripheral activity is said to be **on-line** when it is **directly** controlled by the computer system.

You've seen how some peripheral transfers of data are very many times slower than others. To make the most of microsecond **processing** speeds, you might decide to write **all** your computer output to magnetic-tape.

You might then take your output reels of magnetic-tape to a quite separate computer to get the data printed. This is called **off-line** printing.

Any device which translates data into another medium—say, cards to tape, or tape to print—and which is quite separate from the **processing** computer. is said to be **off-line**.

Off-line working is one way—an expensive way—of running a computer installation at the speed of the fastest peripherals. Even so, it takes no account of the even faster processing speeds inside the central processor of which even the fastest peripherals fail to take full advantage.

The ICT Series 1900 computer system has a much better answer which you'll meet later in this course. However, the terms **on-line** and **off-line** are useful to know.

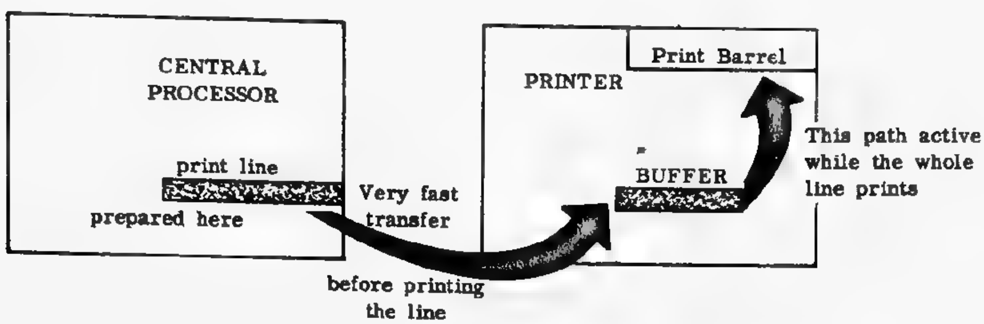
You may think that this pre-occupation with speeds is unnecessary, especially when all the peripherals seem pretty fast anyway. Why can't everybody be satisfied with a flow of data which depends on line printing at 1350 lines a minute?

Again, you'll find an answer to this later in the course. For the present, there's the simple fact that if one way of working is forty times faster than another, this means forty days' work in one day! Or, the work of forty computers on one computer!

The I.C.T. 1933 Printer is an **on-line** printer, then. In fact you could call it an on-line line printer!

It's also **buffered**. This means it has a special intermediate station to which the line of characters to be printed are transferred from the central processor.

The line of characters are printed from this intermediate station whilst the central processor is free to prepare the next line.



Can you see what would happen if you had no print buffer?

Exactly! If there's no buffer, your print line in the central processor transfers direct to the print barrel and you must wait until one line is completely printed before you can **begin** to assemble the next line of characters.

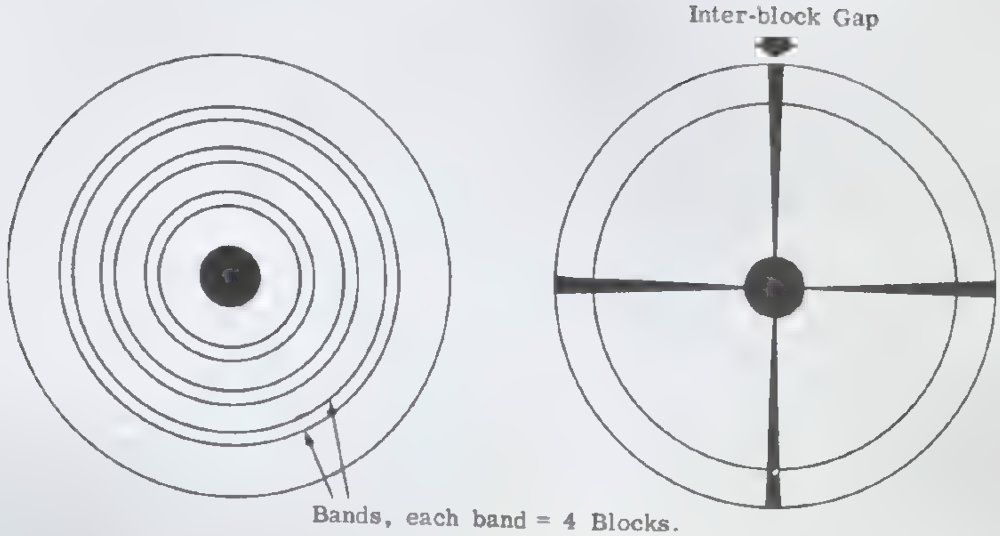
THE I.C.T. 1953 EXCHANGEABLE DISC STORE

Magnetic-tape, paper-tape and punched cards are not the only media for presenting input data and accepting output data.

The I.C.T. 1953 Exchangeable Disc Store is another input and output peripheral. It can supply data to the central processor as input, and accept data from the central processor as output.

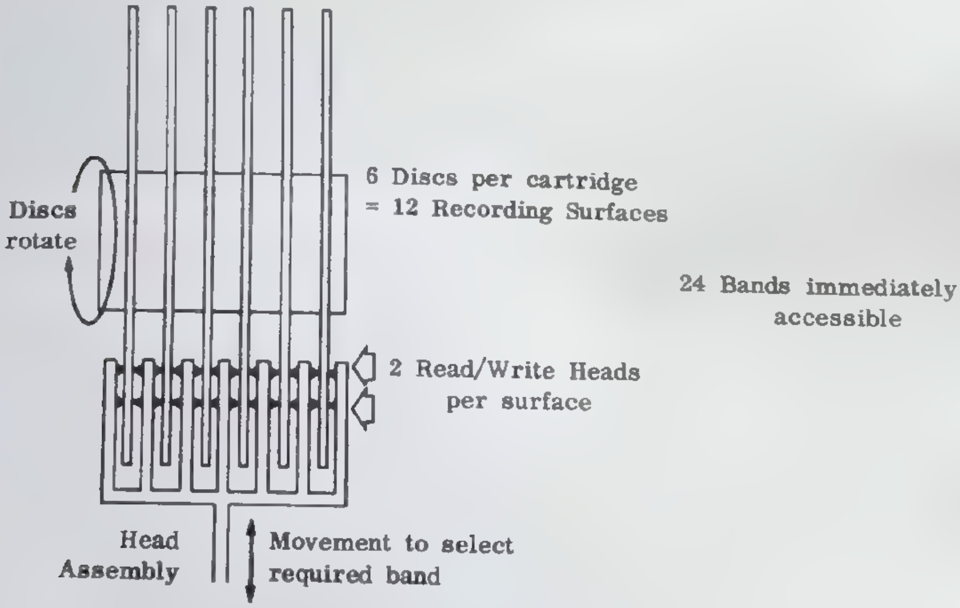
The data is held on what can best be likened to gramophone records—discs, on each surface of which are 200 bands. Each band can hold 1,680 characters in a code of magnetised spots.

This is how a disc surface is divided up:



1 Disc surface has 200 bands which can hold 336,000 characters.

And this is how the data is written to and read from the six discs in the cartridge:



Six discs are contained in a removable cartridge. The picture on the previous page shows the cartridge partially inserted.

Like cards and both sorts of tape, this form of data recording has unlimited capacity—an installation may have any number of disc cartridges. The great advantage is the ease of access to any data in the cartridge; and the characters are read or written at 66,000 a second.

The exchangeable disc store is an extremely convenient medium for holding master-files. It's as fast as all but the fastest magnetic-tape units; it is even more convenient and simple to handle and store; and it has this added advantage of access to **any** data—not merely the next in sequence.

This sort of access to data is sometimes called **random access**. The data may be filed in order, of course, like any manual office file, but access to any item does **not** depend on working through all the file in sequence.

Other Input and Output Devices

You have now met a variety of devices for getting digital data into the computer's central processor and for receiving the central processor's output. Let's briefly review the more important of the devices we've not yet mentioned.

You've met punched card **reading** as input. Cards can be **punched** with output data on a peripheral Card Punch Unit at rates of up to 450 characters a second.

Paper-tape punching is even slower—up to 110 characters a second.

Slower still—but very useful—is the enquiry or interrogating typewriter which is used manually to enter messages into the central processor and which receives automatically typed replies. Its speed limit is 10 characters a second.

Other **high**-speed devices include data disc files which employ permanently mounted recording discs similar to those in the exchangeable disc cartridges. The ICT 1956 Data Disc Store holds 31,500,000 characters. It has two character rates—100,000 characters a second or 60,000 characters a second according to the position of the data.

This is a very important device which allows random access to an enormous store of records.

We've chosen to regard all peripheral units as input/output devices which, indeed, they are, but devices like the Data Disc Store are commonly called **BACKING** or **SECOND LEVEL** STORAGE—because they hold files of data which is permanently 'on tap', as it were.

Another advanced form of input and output, **magnetic card files**, has even greater holding capacity and a character rate of 80,000 characters a second.

Finally in this section, here is a list of typical input/output speeds for an advanced computer:

Slow-speed media

Input (which can be originated by hand)

Card reading at 1,000 characters a second.

Paper-tape reading at 1,000 characters a second.

Interrogating typewriter messages at up to 10 characters a second.

Output

Card punching at 450 characters a second.

Paper-tape punching at 110 characters a second.

Interrogating typewriter replies at 10 characters a second.

Line printing at, say, 2,000 characters a second.

High-speed media

Magnetic-tape reading (input) and writing (output) at 96,000 characters a second.

Exchangeable disc reading and writing at 66,000 characters a second.

Data disc reading and writing at 100,000 characters a second.

Magnetic card reading and writing at 88,000 characters a second.

CHECKPOINT

Insert the appropriate numbers and words to complete the following statements:

1. 

2. Input and output devices are called ________.

3. High-speed input and output media must be read and written to by ______.

4. An item of data held in a group of adjacent punched card columns is called a ________.

5. Punched cards and paper-tape can be read by computer at speeds of up to ______ characters per ______.

6. Punched cards and paper-tape are ____-speed input/output media.

7. Magnetic-tape can be read and written to by computer at speeds of up to ____ __________ __ _______.

8. Magnetic-tape is a ____-speed input/output media.

9. A computer's printer is usually a _______ printer.

10. An intermediate station which serves as a 'waiting-room' for data is called a ________.

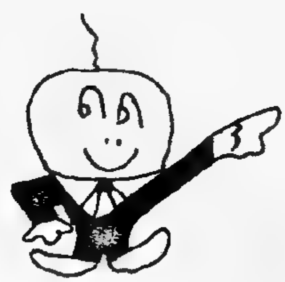
11. Input and output devices connected to the processing computer are said to be __-____.

12. Computer processing speed is ______ than the fastest input and output speeds.

CHECKPOINT ANSWERS

1. Central Processor.
2. Peripherals.
3. Computer.
4. Field.
5. 1,000 second.
6. Slow.
7. 96,000 characters a second.
8. High.
9. Line.
10. Buffer.
11. On-line.
12. Faster.

If you scored 10 or more,



You're ready to start the next section

If you scored less than 10,

You need to look at this last section again

SECTION III

THE CENTRAL PROCESSOR

The main feature of the central processor is **STORAGE**, and the principal idea in computer processing is what we'll call **THE STORAGE CONCEPT**.

The bulk of a computer's central processing unit comprises a STORAGE medium in which data can be held and manipulated.

Now, just consider how simple and essential a concept this is.

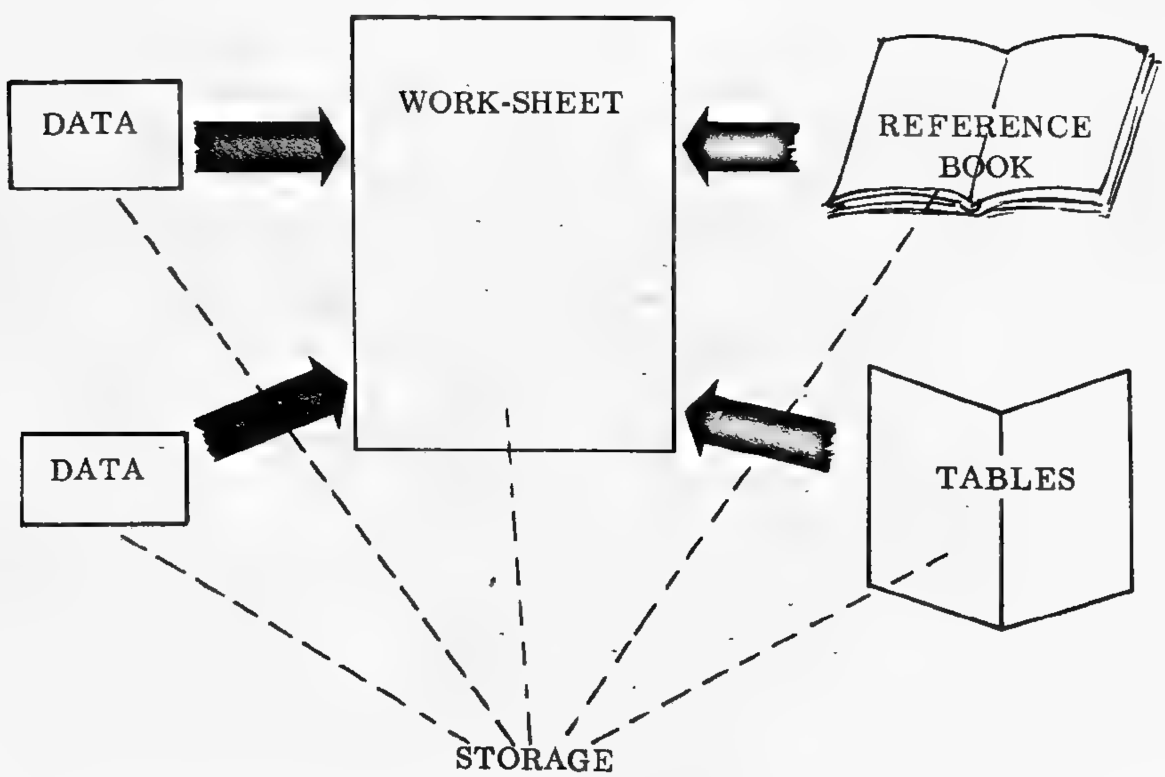
In ordinary, non-computer work, you'd not begin to tackle a calculation of any size without first jotting down the data factors on paper. **THAT'S STORAGE!** After all, you've written down the factors to hold them other than in your head—to **store** them on the sheet of paper.

You would go on to manipulate the factors on your sheet of paper. A computer goes on to do precisely the same in the central processor's STORAGE.

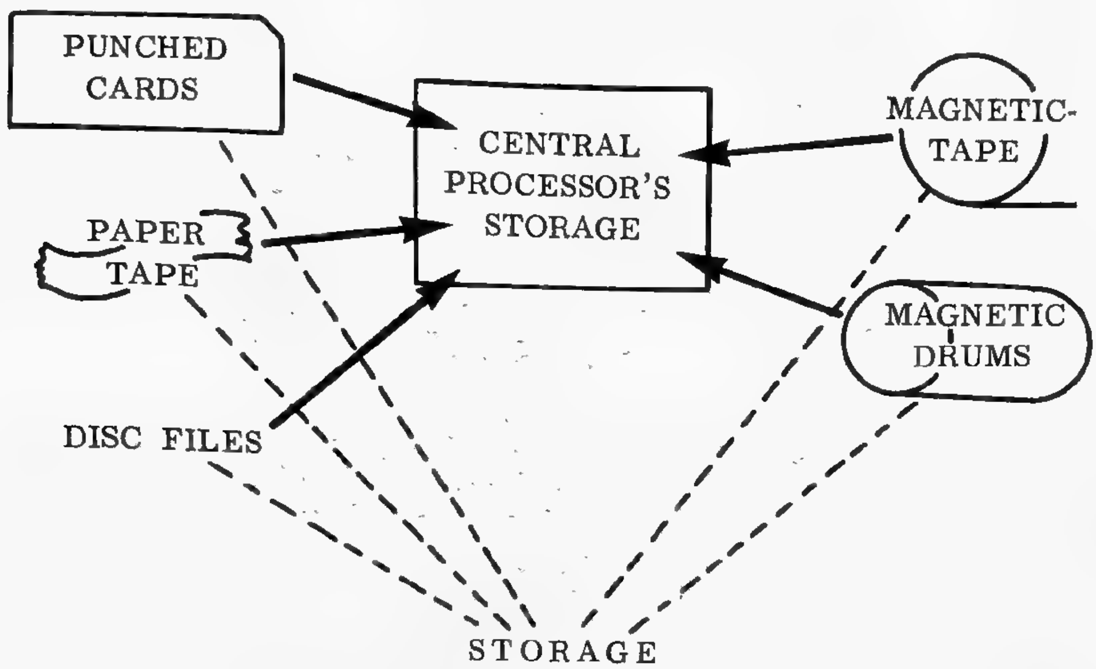
It's almost impossible to strain this analogy between a paper work-sheet in manual work and STORAGE in a computer's central processor.

The trouble is not so much the analogy as the storage concept itself which expands beyond the central processor to include the input and output media.

Thus, if you talk about data factors being **stored** and manipulated on a paper work-sheet, then you must face the fact that any data on any document—or in any book or file—is **stored**:



But this use of the word STORAGE really belongs to computers:



Clearly, WE WANT A **SPECIAL** NAME FOR OUR 'WORK-SHEET' STORAGE—THE CENTRAL PROCESSOR'S STORAGE.

Unfortunately, a variety of terms have gained at least **some** currency. Here they are:

Immediate Access Storage (I.A.S.)
Fast Access Storage
Central Store
Working Store
First Level Storage
Core Storage
Magnetic Core Storage
Magnetic Ferrite Core Storage

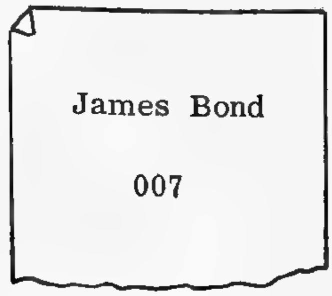
Those last three terms. CORE STORAGE, MAGNETIC CORE STORAGE, MAGNETIC FERRITE CORE STORAGE, are all based on the same physical feature.

This is the storage medium usually used in the central processor of a modern digital computer.

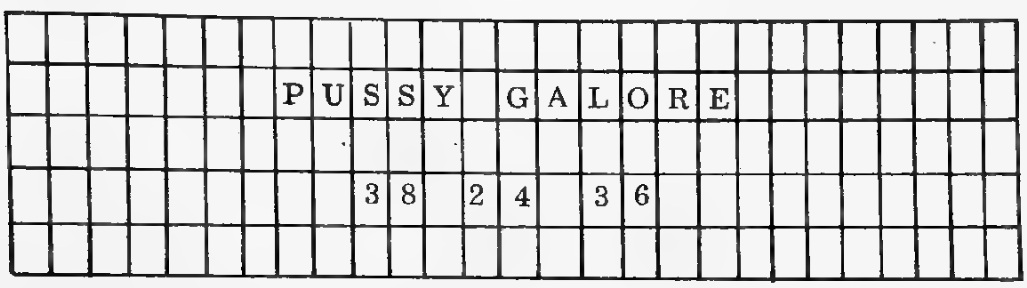
Inevitably, magnetic ferrite core storage will eventually be superseded by an even faster, even less bulky, even more generally convenient form of storage. But at the time of writing, it seems set fair to continue for many years as the best fast-working medium for the central processor.

So, let's call our 'work-sheet', the central processor's store, **CORE STORAGE**. As a name, it not only honours the ferrite cores, it also suggests the 'central' idea. It's short, too.

Just as you can record data characters on paper with pen, pencil or type,



so CORE STORAGE can hold data characters ready for processing:



You'll no doubt appreciate that the computer programmer may use a **core storage chart** of squared paper on which, as he writes the computer program, he keeps track of the factors and the processing—in ordinary character form.

Let's summarise this central idea:

* To tackle a calculation of any size, in the ordinary way you'd use a sheet of paper to record the data factors, the various stages of your working and the final results, before presenting them in the required form.

* Within its central processor, your computer has a central store in which data can be held and manipulated in much the same way as on a paper work-sheet.

* Physically, of course, the central store has nothing in common with a sheet of paper. Most commonly, it is made up of magnetic ferrite cores and we've decided to call it CORE STORAGE.

* The computer programmer makes use of a squared-paper representation of CORE STORAGE—a core storage chart.

If core storage always held data in CHARACTER FORM, then you'd never need to trouble your head with the means of holding the characters on ferrite cores.

You'd know that each square on your squared-paper could hold a numerical digit, an alphabetical letter, or a special symbol (/, *, +, etc.) and this would be enough. How the computer managed to record the characters would be of passing interest only.

However, some computers use the **binary** rather than the decimal number system, so we ought to take a quick look at each in turn.

Let's look first at a method of holding **decimal** and alphabetical characters.

Let's consider a simple code for numbers and letters.

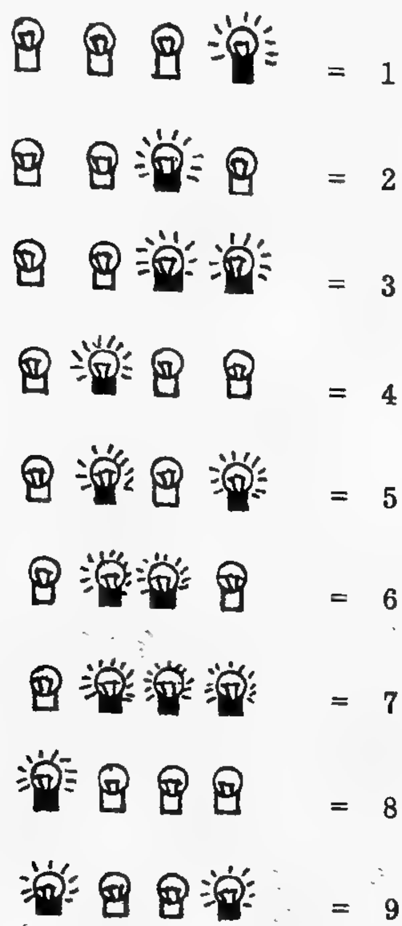
You could have a primitive sort of code with, say, dots only, (unlike the Morse code, for example, which has dots and dashes):

.	=	1
. .	=	2
. . .	=	3
. . . .	=	4
. . . . .	=	5
. . . . . .	=	6
. . . .	=	7
. . . .	=	8
. . . .	=	9

But this is a very clumsy code. One decimal digit needs up to 9 indicators.

Here's a much neater idea:

For each decimal digit, always **four** indicator positions are used, but each position may be **ON** or **OFF**:



In Core storage, the ON/OFF devices are ferrite cores, but electric lamps illustrated the principle exactly.

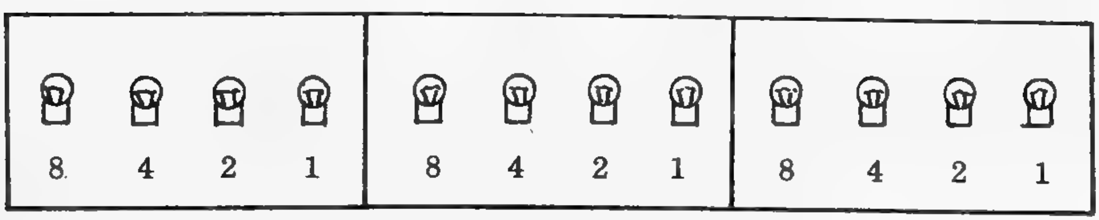
The separate indicators in a computer code system, whether they're ferrite cores or flashlight bulbs or whatever, are called **BITS**.

In our example above, what value have we given to each of the four BITS? Write down your answer before going on.

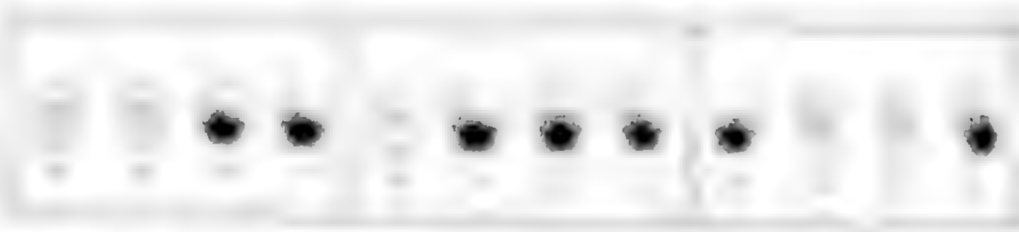
You were right? Good!

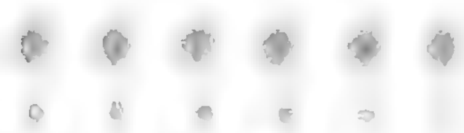
Now remember, this is for recording **decimal** characters. Each character position needs four ON/OFF bits.

How will these three character positions hold the number 379?



Mark your copy to show which lamps should be ON.





The letters A to I can be represented by the decimal digits 1 to 9 **plus** the X bit, the letters J to R by 1 to 9 **plus** the Y bit, and the remaining eight letters by 2 to 9 **plus both** the X and Y bits.

Write down this code's version of the letter V.

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

101001	100011	110011	000001	001001	000000	000011
I	C	T	1	9	0	3

01011

If you failed to understand what was required, go back and read the last two pages again.

Typically, Core Storage consists of several thousands of these multi-core locations.

(Apart from holding data factors, constants, tables, and providing working space, there's another very important reason why Core Storage must be sizeable. We'll come to it presently.)

You've seen that this character code—called BINARY CODED DECIMAL (BCD)—is based on bits which can be in one of two states, on or off—BINARY BITS.

Not surprisingly, therefore, some computers have the ferrite cores grouped for **pure** BINARY ARITHMETIC instead of BCD.

Binary arithmetic is very simple.

Decimal work takes the base 10. You hold no more than 9 in each position and, in excess of 9, a 'carry' forms to increase the next higher position. This is the ordinary number system we're all familiar with.

Instead of base 10, BINARY ARITHMETIC takes the base 2. You hold only 1 or 0 in each position and, in excess of 1, a 'carry' forms to increase the next higher position. So, instead of ten thousands, thousands, hundreds, tens and units, you have:-

e.g.,

16's	8's	4's	2's	Units	
1	1	0	1	1	= 27

In representing each **decimal** digit in the BCD code, you use the binary system—but only the bottom four positions of the binary system:

8's	4's	2's	1's	8's	4's	2's	1's	8's	4's	2's	1's	
0	1	0	0	0	0	1	0	0	1	0	1	= 425

The same number in pure binary looks like this:

256's	128's	64's	32's	16's	8's	4's	2's	1's	
1	1	0	1	0	1	0	0	1	= 425

Compare the number of bits used in the two systems.

Twelve bits were needed to hold 425 in BCD; in pure binary, only nine.

And remember, BCD normally uses not four, but **six** bits per character (XY8421).

The comparison is thus eighteen bits as against nine.

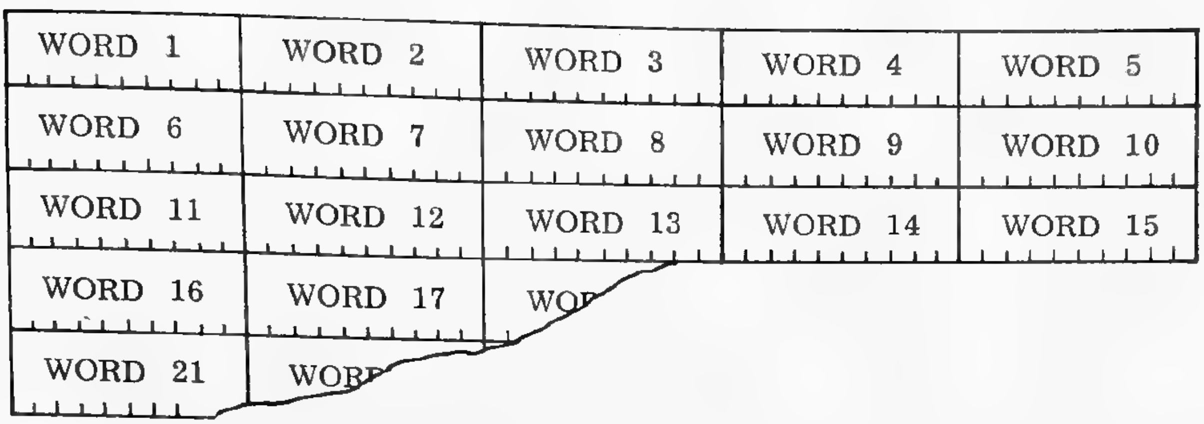
You can see that pure binary is simple and economical. (It not only saves expensive ferrite cores, it saves computer processing time too).

Some digital computers are pure binary machines throughout and are not designed to hold alphabetical and decimal characters. This is fine for such scientific work as requires only numeric answers, but no good for business applications which require so much character data to go through the computer unchanged.

The ICT Series 1900 offers the best of both worlds. Arithmetic is pure binary, but for non-arithmetic purposes the computer will regard six adjacent binary bits as a BCD character.

Some computers have their core storage divided into **WORDS**—compartments comprising 8, 12, 24, or some other convenient number of either character locations or binary bits.

CORE STORAGE CHART



A data factor will thus be stored in a WORD. Factor A may be stored in WORD 9 and factor B in, say, WORD 12.

What, in your opinion, would be the **ADDRESS** of factor A and what the **ADDRESS** of factor B?

Write down your answer before continuing.

You should have chosen '9' or 'WORD 9' as the ADDRESS of factor A, and '12' or 'WORD 12' for the ADDRESS of B.

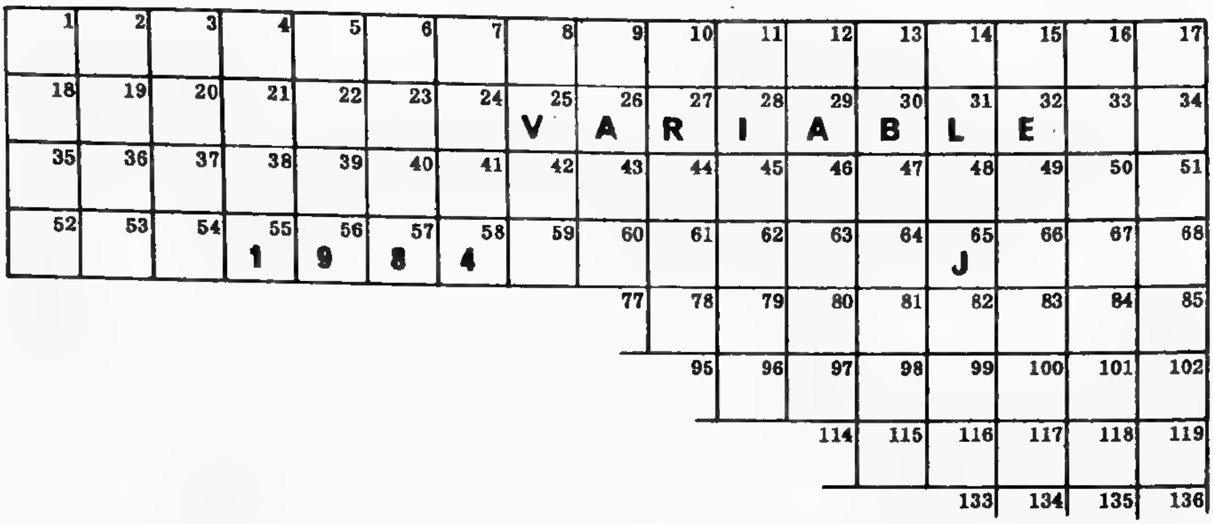
The computer must be told where to go to find any given item of data. Data in core storage must be **ADDRESSABLE**. and a system of words with word numbers is clearly a system of **ADDRESSES**.

A computer which has its core storage divided into these distinct and separate words is called a **FIXED WORD-LENGTH** machine.

Such machines are sometimes designed so that **HALF-WORDS** may be dealt with separately or so that two words may be treated as one for **DOUBLE-LENGTH WORKING**.

But the principal fact is **ADDRESSABLE WORDS**, each with a fixed number of characters or binary bits.

The alternative system gives an **ADDRESS** to each character location.



You'll guess the ADDRESS of the data item 'J', at once. Yes, it's obviously 65.

But what of the items 'VARIABLE' and '1984'?

'VARIABLE''s address will be either 25 or 32 depending on the design of the computer. Similarly, '1984' is found at either 55 or 58, again, according to the specific design.

To make this system work, the computer must also be told . . . . . .What?

Answer: The limit of the word—in other words, when to stop processing that word of data. You'll see how this might be done presently.

Computers with this sort of Core Storage are called **VARIABLE WORD-LENGTH** machines—for the obvious reason.

They're also called **CHARACTER** machines—for a double-barreled reason. First, each **character** is addressable, and second, this sort of computer invariably processes characters rather than using pure binary arithmetic.

In a variable word-length computer, whatever the size of the data field, the storage word fits it exactly.

Earlier, we remarked that Core Storage must be sizeable not only because it must hold data factors, constants, possibly tables and any other sort of data which might be required in processing records, but for another important reason.

Core Storage also holds the **PROGRAM** of instructions. This, indeed, is one of the most important characteristics of a computer—CORE STORAGE HOLDS BOTH THE DATA **AND** THE PROGRAM OF INSTRUCTIONS.

A highly complicated program of instructions will occupy thousands of character locations. Don't be surprised, therefore, to come across computers with Core Storage running to tens of thousands of character locations—which may mean the best part of a million ferrite cores.

Look at the following core storage chart very carefully. It shows a mass of alphabetic and numeric characters held in core storage. Both data factors and program instructions are shown here.

Can you see where the program begins?

1: 6	2: 3	3: 4	4: A	5: 4	6: 5	7: 9	8: 6	9: 8	10: 2	11: 9	12: 5	13: 7	14: C	15: H	16: 2	17: B
18: 1	19: 3	20: 3	21: S	22: 4	23: 8	24: 5	25: 9	26: 7	27: 2	28: 8	29: 3	30: 1	31: 2	32: 7	33: 9	34: M
35: 2	36: 4	37: 5	38: 9	39: 8	40: 7	41: 2	42: 4	43: 5	44: 9	45: X	46: 1	47: 2	48: 4	49: T	50: W	51: O
52: 4	53: 5	54: 9	55: 8	56: 4	57: 2	58: 6	59: 4	60: 4	61: D	62: Y	63: 0	64: 2	65: 2	66: 3	67: 4	68: 5
69: 1	70: 6	71: 4	72: 5	73: 7	74: 2	75: 4	76: 9	77: 8	78: 1	79: C	80: 2	81: 7	82: 5	83: 9	84: 5	85: 8
86: D	87: 5	88: 3	89: 7	90: 9	91: 4	92: 2	93: 2	94: 6	95: 2	96: 1	97: B	98: 8	99: 9	100: 3	101: 2	102: 6
103: 2	104: 7	105: 4	106: 3	107: 6	108: 9	109: 1	110: 1	111: 6	112: 5	113: 2	114: 4	115: 3	116: 8	117: M	118: 1	119: 5
120: 9	121: 8	122: 4	123: 3	124: 6	125: 5	126: 2	127: 1	128: 3	129: 6	130: 8	131: 6	132: 4	133: 3	134: 1	135: 5	136: 8
137: 9	138: S	139: 4	140: 6	141: 8	142: 5	143: 6	144: 3	145: 9	146: 4	147: 7	148: 1	149: 4	150: 8	151: 4	152: F	153: 1
154: 8	155: 6	156: 3	157: 2	158: 1	159: 7	160: N	161: 5	162: 8	163: 7	164: 5	165: 6	166: 3	167: 4	168: 2	169: 2	170: 9

Be brave, and commit your answer, 'Yes' or 'No', to paper before you go on.

Unless you have strange powers of divination, your answer must be 'No'. Without knowing the specific computer, (which may have some convention about where the program is stored), you can't tell which of these characters belong to data factors and which to instructions.

Program instructions are made up of characters or binary numbers—just like data, in fact.

An instruction has a pattern called the INSTRUCTION FORMAT.

A typical instruction format for a **variable** word-length machine might contain eight characters, like this:

3	0	4	4	0	7	5	4
The Operation Code	The A Address			The B Address			Extra Character

You can probably guess what some parts of the instruction represent.

3	0	4	4	0	7	5	4
The Operation Code	The A Address			The B Address			Extra Character

The OPERATION CODE represents the operation to be carried out—adding, subtracting, moving a factor from one address to another, reading in a record from some input device, writing a record to some output device, multiplying, dividing, and so on, (e.g., 3 = ADD).

It doesn't matter that the operation code is an ordinary character which may well occur elsewhere in the instructions or in the data.

The computer can recognise the OPERATION CODE for what it is and act accordingly because:-

1. The computer alternates continuously between a) analysing the next instruction, and b) obeying that instruction.

   A computer is designed to work in a) instruction phase, b) execution phase, a) instruction phase, b) execution phase, and so on. You probably work in precisely the same way when you carry out a series of tasks by following a list of instructions.

2. The computer goes to work on an instruction from left to right. In the instruction phase, the OPERATION CODE will thus be the first character it encounters.

The A Address and the B Address of the instruction tell the computer where to go to find the two data factors.

Some computers use this two-address system principally because many operations naturally have two factors—for example, subtraction, multiplying, dividing, comparing, moving data from one address to another.

Other computers use a **one**-address system and deal with one factor at a time.

Yet others have a **three**-address system which can place the result in a different address from those of the two factors.

Each system may have advantages according to design.

3	0	4	4	0	7	5	4
The Operation Code	The A Address			The B Address			Extra Character

Let's look at our example. First let's see what this instruction does.

The factor at the A address is added to that at the B address. The answer replaces factor B. (If you want to keep it, you store factor B elsewhere as well as at the B address.)

BEFORE

FACTOR A

041	042	043	044
0	2	3	4

FACTOR B

071	072	073	074	075
	1	4	7	2

AFTER

FACTOR A

041	042	043	044
0	2	3	4

A + B

071	072	073	074	075
	1	7	0	6

Our example is based on the type of computer which adds the two factors together digit by digit starting at the rightmost locations, 044 and 075. (You probably add two numbers in precisely the same way.)

The arithmetic unit in this type of computer adds only one pair of digits at a time in what's called a **one-digit-adder**.

Because the answer replaces factor B in Core Storage rather than in a special accumulator this is often called an **Add-to-storage** or **Add-to-memory** machine.

Consider what happens in each of the two phases, instruction phase and execution phase.

In instruction phase, the instruction characters are moved to a special set of ferrite core positions. There, their actual code pattern is tested electronically in such a way that, as a result, the required circuits are set-up.

Testing the Operation Code, 3, standing as it does in the first of these special core positions, 'makes' the circuits which will set in motion the adding process.

Testing the A-Address cores 'makes' the circuit which connects the single location, 044, to the one-digit adder.

Testing the B-Address cores does the same for location 075.

The Extra Character, 4, is held so that it can be decremented by 1 as each pair of digits are added. When it becomes 0, the operation will stop and a four-digit addition will be complete.

When all eight characters of the instruction have been moved into these special positions, the computer switches over to execution phase.

In execution phase, the operation is carried out.

First, the digits at 044 and 075 are added in the one-digit-adder and the result stored at 075. A 'carry' to the next position would be held in the adder. The Extra Character, 4, is reduced to 3.

Then, 043 and 074 are connected to the adder and their digits are added together with any 'carry' from the first pair. A 'carry' to the next position is held in the adder. The Extra Character is reduced to 2.

The same procedure follows for 042 and 073 and the Extra Character is reduced to 1.

Finally, 041 and 072 are added together and the Extra Character is reduced to 0.

This stops the operation and switches the computer to instruction phase on the next instruction.

The computer does this all the time—'analyses' an instruction, then obeys it, 'analyses' the next instruction, then obeys it—an invariable routine.

The actual mechanisms which scan core storage to keep track of both the program and the data factors need not concern you.

On the programming side you are concerned with the logical rather than the electronic and electromechanical, aspect of computers.

Given the rules of what can happen to instructions and data within a specific computer, we accept them and stick to our numbers and letters on squared-paper.

At some time or other, the programmer will want to trace through the circuitry which allows these effects, if only out of curiosity.

Beginners are sometimes confused by this business of 'understanding' the computer and writing program instructions without knowing the full electronic story. But we all accept this sort of situation in more familiar spheres without a moment's hesitation.

You drive a motor-car by giving 'instructions' to the gear-box, the engine, the clutch, the steering mechanism. These 'instructions' are in the form of simple hand and foot operations. You can perform these operations with complete efficiency even though you may be totally ignorant of how the effects are achieved in mechanical detail.

The girl on the office telephone switchboard knows how to cause all the effects she requires. She probably knows little of what happens between her switchboard and the various exchanges and office telephones.

Try writing an ADD instruction yourself.

65,983 is a total to which you want to add 12,492. Remember, the Operation Code for Add is '3'. Here are the two factors in Core Storage.

116	117	118	119	120	121	122	123
		6	5	9	8	3	

200	201	202	203	204	205	206
	1	2	4	9	2	

Simply write the eight-character instruction which will add the 12,492 to the 65,983, leaving 12,492 as it stands and the new total in place of 65,983.

Turn back to page 26 if you need a reminder but don't go on until you've had a shot at it.

The answer: | 3 | 2 | 0 | 5 | 1 | 2 | 2 | 5 |

If you got it, well done! If not, look at the last few pages again to see where you went wrong.

So, the computer programmer writes precise instructions in a set format and these instructions must be fed into the computer before the actual data processing job can begin.

The program of instructions will occupy part of Core Storage just as some other part of Core Storage will hold the data factors.

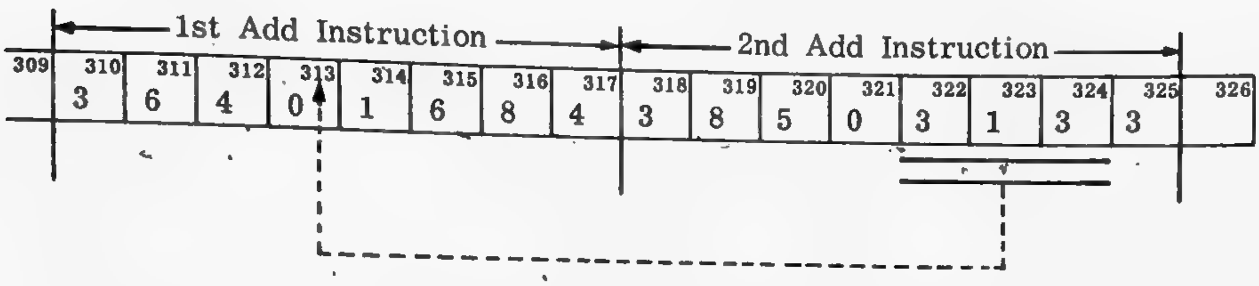
Program instructions are made up of characters or binary numbers—just like data.

THE PROGRAM CAN, THEREFORE, PROCESS ITS OWN INSTRUCTIONS JUST AS IF THEY WERE DATA WORDS

This is quite a thought ....

. . . . and, when you think about it, almost an obvious one.

Program instructions are held in Core Storage just like data words, so you can add to or subtract from, or otherwise modify instructions just as you can data.



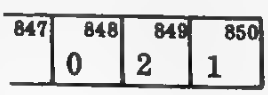
Can you see that the second Add instruction modifies the first by simply adding to the A Address, 640?

It does this by addressing 640, just as if it were a data factor, at its address, 313.

It adds a factor which has been positioned at address 850.

Let's see if you've managed to sort this out.

At 850 we've stored 21 like this:



Now, assume that the second add instruction has been obeyed, and write down the first instruction in its modified form.

After the first instruction has been obeyed in its original form, and after the second—the modifying instruction—has been obeyed, the first instruction will look like this:

309	310	311	312	313	314	315	316	317
	3	6	6	1	1	6	8	4

If your answer is incorrect, turn back and work through this rather tricky argument again.

This process can go on changing the A Address in the first instruction with the same factor at 850 for every pass of the program.

Or, another instruction might supply a **varying** factor at 850.

You might even modify the modifying instruction!

In these and many other ways you can build into the program the sort of 'self-change' which allows the computer to take the appropriate alternative action as factors and circumstances change and develop.

You'll meet other simple examples of INSTRUCTION MODIFICATION later in the course. Our point at the moment is the tremendous general significance of the concept.

Instructions can command all the basic arithmetic operations and the recognition and movement of data. They can be sequenced and repeated in an infinite variety of ways. Quite literally, the processing possibilities are unlimited—except, of course, by the sheer size limitation of Core Storage.

And, because instruction words are held just like data words, INSTRUCTION MODIFICATION—the program's capacity to change itself—is similarly unlimited.

We said earlier that one of the most important characteristics of a digital computer was the stored program.

Instruction modification is another.

Some people insist that these are the two characteristics which distinguish a computer from other automatic data processing machines like punched card tabulators and calculators.

These less powerful machines are also controlled by a series of instructions, and some have a very extensive range of alternative action.

But this facility for **unlimited** instruction modification belongs solely to **stored** program computers, and this almost brain-like power to adapt action to suit circumstances goes some way towards justifying the popular term, ‘electronic brain’.

You may still be wondering how you know where to place data factors in core storage and where instructions.

For some computers, the programmer can choose for himself, almost without restriction. Practice is as uncomplicated as principle. The programmer decides on what seems to be the most productive arrangement for the job in hand.

Other computers have special areas of core storage set aside for this and that—to accept incoming data or as setting-up areas in which to arrange output formats, and so on. Fixed word-length machines invariably have special words, **ACCUMULATORS**, to hold all arithmetic results. But there's no real difficulty here: the programmer uses the rest of core storage to suit himself.

In the most advanced computers, as you'll see in a moment, a specially prepared 'master' program—a sort of resident 'Big Brother' in core storage—master-minds all sorts of decisions including where to start the program. The programmer need never concern himself with the position of either data factors or program instructions.

Let's list the points you've seen since we summarised the central idea of Core Storage.

1. Core Storage holds data in binary form or in some extension of binary form like Binary Coded Decimal. The binary on/off device—the indivisible unit in either system—is called a BIT.

2. A group of **character** locations or **pure binary** locations is called a storage WORD.

3. Computers are either FIXED WORD-LENGTH machines or VARIABLE WORD-LENGTH machines.

4. Data factors must be addressable. Fixed word-length usually allows each word to be addressed. Variable word-length implies an ADDRESS for each character location, and a whole word may be addressed by, say, the address of its right-most location **plus** an indication of the word's length or limit.

(Summary continues ⟶

(Summary continued)

5. Core Storage holds both the data **and** the PROGRAM of instructions. Like a data word, an instruction is made up of numbers or numbers and letters.

6. Instructions are held in a set INSTRUCTION FORMAT: e.g.,

3	0	4	4	0	7	5	4
The Operation Code	The A Address			The B Address			Extra Character

Different computers have different formats and you may meet one-, two- or three-address systems.

7. A computer 'reads' an instruction and then obeys it, 'reads' the next instruction and then obeys it, and so on. When the computer is running a job, this alternation between instruction phase and execution phase is continuous—and, of course, quite automatic.

8. A typical program contains hundreds or even thousands of instructions.

9. The programmer is rarely an expert in electronics. He is an expert in analysing the problem in computer terms and in using the instructions to write a workable program.

(Summary continues →

(Summary continued)

10. Because instructions are held in Core Storage just like data, they too can be processed. This is called INSTRUCTION MODIFICATION. It opens up unlimited possibilities for alternative action.

11. Most computers have some areas of Core Storage reserved for special purposes. Arithmetic results may be held in special words called ACCUMULATORS. Input and output may have special areas. The programmer may be able to choose for himself where in Core Storage to place the data factors and the program.

12. The more advanced computers hold a specially prepared 'master' program in Core Storage to carry out many functions which would otherwise require much more programming by the programmer. This 'master' program will normally allocate Core Storage areas for data and for the program.

Now, to round off this section on the central processor, we follow up point 12 and at the same time take a closer look at the I.C.T. 1900.

So far, we've inclined towards the variable word-length, add-to-storage, character machine—such a system being perhaps the easiest to think about.

But avoid making the equation,

$$\frac{\text{MOST EASILY DESCRIBED}}{\text{CORE STORAGE SYSTEM}} = \text{BEST COMPUTER.}$$

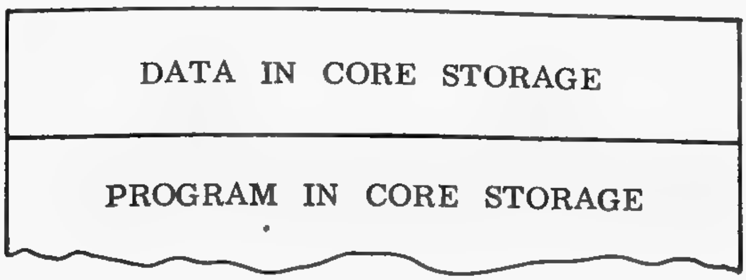
It's very tempting—for the beginner especially—but it's not true.

Let's look now at the storage word of a **fixed** word-length machine which uses both the pure binary and the character system.

Because our example is an advanced computer, things get a bit more complicated. Don't worry. We'll hold on to a simple, untangled life-line.

Like good sailors, let's rig the life-line, first:

When you have,



you have the essential digital computer set-up.

So, although you may be presented with more than this, or with this several times over, you can find these elements and say to yourself, "Ah! Data and program in Core Storage! In an ordinary, simple computer this would be the lot."

An I.C.T. 1900 computer system may contain a central processor which is capable of switching from one to another of several programs. The master program which is called into play to monitor all this is called **EXECUTIVE.**

CORE STORAGE

EXECUTIVE		
PROGRAM A	-	DATA
PROGRAM A	-	INSTRUCTIONS
PROGRAM B	-	DATA
PROGRAM B	-	INSTRUCTIONS
PROGRAM C	-	DATA
PROGRAM C	-	INSTRUCTIONS
PROGRAM D	-	DATA
PROGRAM D	-	INSTRUCTIONS

Take a grip on that life-line. In a less advanced computer's Core Storage, any one of these programs with its data factors would make up the total picture.

INCIDENTALLY,

do you remember what we said in Section II about the speed of processing compared with the speed of input and output operations?

Fast as high-speed peripherals are, they're still far too slow to match microsecond **processing** speeds.

A partial solution, you remember, was to use high-speed peripherals only, and use off-line devices for 'translation'.

We promised a **real** solution. This is it—**MULTIPROGRAMMING.**

During the comparatively lengthy peripheral operations, the computer can carry on with the processing **for more than one program**.

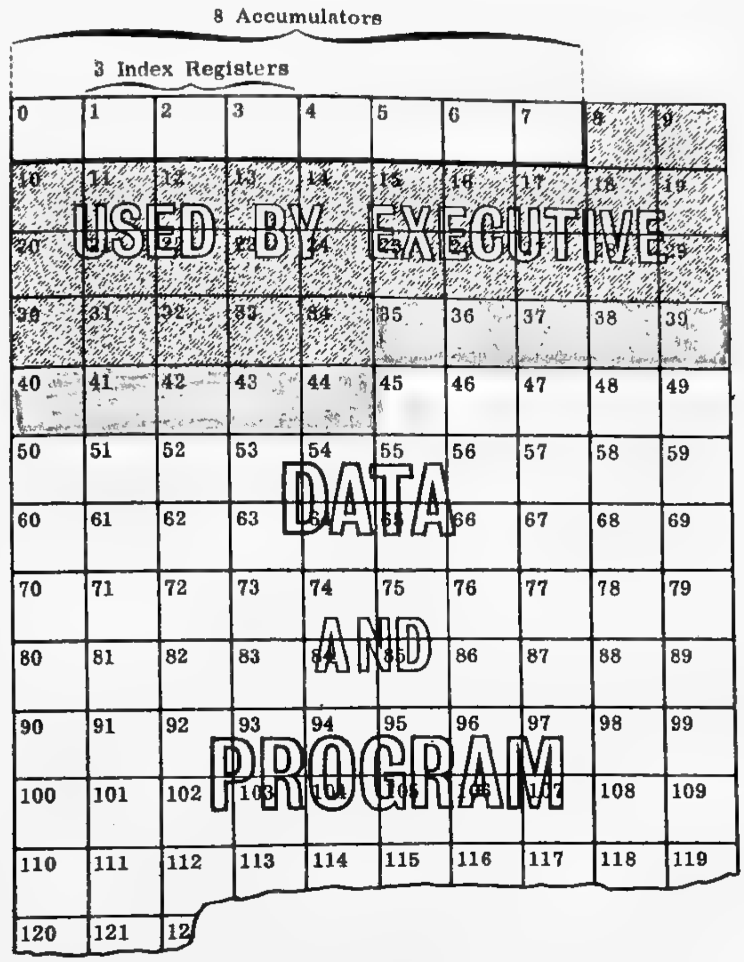
Executive monitors the whole business.

Another rather cunning aspect of this system—only Executive knows where the program and the various data factors are. The allocation of core storage is carried out entirely by Executive. And Executive can eliminate an unwanted program and reshuffle the remaining programs!

(This is fantastic, isn't it? But, in fact, there's nothing magical about it. Executive is just a series of **SUB-ROUTINES** (bits of program) which have been written by programmers—albeit expert programmers—using the ordinary instructions to which the computer is designed to respond. Remember, we decided earlier that there's no limit to a computer's processing and instruction modification capabilities—except that imposed by size, of course.)

So, since whole programs can be moved to new pastures in core storage, let's consider **one program only**, for a moment. Let's assume it doesn't move at all, and that it occupies the very beginning of core storage, onwards, starting at word 0.

This is how each program's area begins:



Each box is a 24-bit word which may be used in a variety of ways.

The programmer can regard each program's Core Storage area in this way. But this is a polite fiction, of course. Throughout Core Storage, each address must be quite different from any other. Words are, in fact, numbered serially from the beginning to the end of Core Storage as in any other computer.

Executive translates these fictitious addresses for each program into actual machine addresses and so allocates a different area of Core Storage for each program.

For each program, then, the first eight words, 0 to 7 are ACCUMULATORS. These are arithmetic registers into which the contents of other words may be added or subtracted. Multiplication and division are also effected in accumulators.

The contents of an accumulator may be tested to reveal results of zero, non-zero, greater or equal to zero, and less than zero.

Accumulators are also used for three binary logic operations which need not concern us here, the 'logical and', the 'logical or', and the 'exclusive or'. The purpose of these is to change the pattern of binary bits in an accumulator on the basis of a bit-by-bit comparison with another word.

Accumulators 1, 2 and 3 have a further special function as INDEX REGISTERS. They can be used to hold constants which will effect automatic **INSTRUCTION MODIFICATION** by automatically modifying an address in the instruction word.

An Index Register's constant may also be used as a COUNT to effect an automatic exit from a program loop after that number of passes. But more of that in the next section.

Basically, each word in the I.C.T. 1900's Core Storage is exactly similar to any other word. The eight accumulators for each program have special access to the arithmetic circuitry, but otherwise they too are exactly similar to the other words.

Remember our life-line: essentially, Core Storage holds data words and instructions.

First, let's look at the 1900 core storage word as a data holder.

Apart from a 25th bit which the computer uses in an automatic check against machine error, the word comprises 24 bits numbered 0 to 23.

$$\underbrace{\boxed{0}\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5}}_{n_0}\underbrace{\boxed{6}\boxed{7}\boxed{8}\boxed{9}\boxed{10}\boxed{11}}_{n_1}\underbrace{\boxed{12}\boxed{13}\boxed{14}\boxed{15}\boxed{16}\boxed{17}}_{n_2}\underbrace{\boxed{18}\boxed{19}\boxed{20}\boxed{21}\boxed{22}\boxed{23}}_{n_3}$$

Why do you think the 24 binary bits may be regarded as four groups of six bits, n_0, n_1, n_2 and n_3 ?

Each group of six bits can, of course, hold an alphabetic or decimal character, like this:

ONE 24-BIT WORD				ONE 24-BIT WORD				ONE 24-BIT WORD			
101010	100001	100011	101011	010000	101010	101111	101000	101110	110011	101111	101110
J	A	C	K	(SPACE)	J	O	H	N	S	O	N

(N.B. The I.C.T. 1900's alphabetical code is slightly different from the one you saw earlier.)

Numerical data on which no arithmetic processing is required may also be held as BCD characters:

ONE 24-BIT WORD				ONE 24-BIT WORD				ONE 24-BIT WORD			
000111	000100	000010	000101	100001	000001	010000	000110	001001	001000	010000	000011
7	4	2	5	A	1	(SPACE)	6	9	8	(SPACE)	3

Now we see the 24-bit word holding a number in pure binary form. Work out its vale:

000000000000000010100101

000000000000000010100101 = 165.

The word's leftmost bit is the sign indication. 0 in this position indicates positive; 1 indicates negative. So, our answer, 165, is quite correct.

The greatest positive number that can be held in a single word, is

$$011111111111111111111111 = 2^{23} - 1 = 8{,}388{,}607.$$

The greatest negative number that can be held in a single word is,

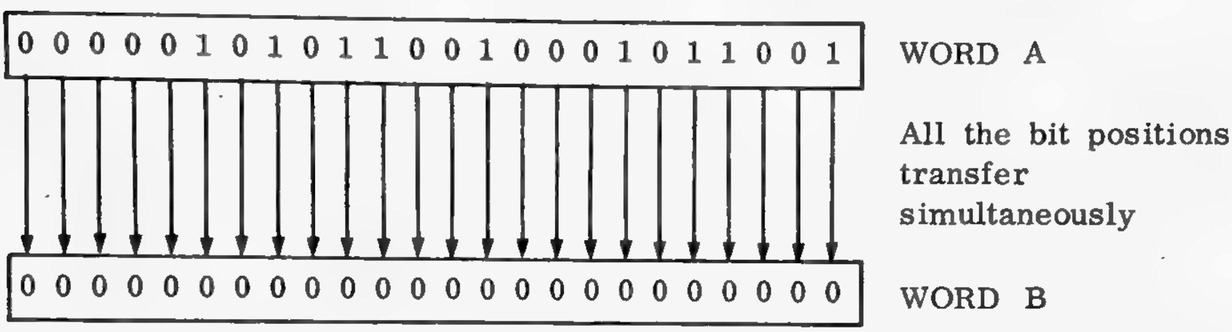
$$100000000000000000000000 = -2^{23} = -8{,}388{,}608.$$

Larger numbers can be held in two or more words. Several small numbers can share a single word. Fractions and mixed numbers (integer and fraction) can be processed with either a fixed or floating decimal point position.

—a pretty comprehensive range of number work which is economical and very fast.

We said earlier that pure binary notation saved ferrite cores. In fact, pure binary numbers occupy up to 40% fewer core storage positions than the same numbers in BCD form.

On top of this, consider the correponding time-saving. The fastest method of bit transfer in computers is the full parallel method:



In the I.C.T. 1903, 1904 and 1905 Central Processors this operation takes 7 **millionths** of a second.

When you remember that the numerical content of a word may be equivalent to 7 decimal digits, this means 1 microsecond per decimal column—a sort of 'even time' for modern digital computers.

Finally in this central processor section, let's look at the I.C.T. 1900's core storage word as it's used to hold program instructions.

Most 1900 program instructions involve the use of an Accumulator. The instruction format, therefore, provides for an Operation Code, an Accumulator address, and the address of a Core Storage word. This last is further provided with a means of address modification.

Normal instructions make use of the 24-bit word like this:

X	F	M	N
3 bits	7 bits	2 bits	12 bits

F contains the Function Code (the 1900 term for OPERATION CODE). The seven bits are divided into three 'columns':

1 bit 3 bits 3 bits

which you can translate into a modified Binary Coded Decimal range of 000 to 177 giving a possible range of 178 different operation codes.

X is the address of one of the eight Accumulators, three bits being sufficient to indicate 0 to 7.

N is the address of a core storage word.

M may be used to modify the address, N.

Do you remember the INDEX REGISTERS? On page 109 you saw that Accumulators 1, 2 and 3 can be used for this purpose. (Automatic address modification is often called INDEXING.)

M, therefore, may be used to indicate that the core storage address, N, should be modified by adding to it the contents of Accumulator 1, 2 or 3. Two bits, as you know, are sufficient to indicate 1, 2 or 3.

The programmer writes 1900 instructions in the order in which we've described them:

F X N (M)

It matters not at all that the computer actually holds the instruction in a different order.

Now we'll look at a couple of simple examples, one which makes no use of M, and one which does.

The 1900's first Function Code, number 000, is 'Load Accumulator'. As its name suggests it simply loads the Accumulator, indicated at X in the instruction, with the contents of the core storage word addressed at N.

F	X	N	(M)
000	5	628	not used

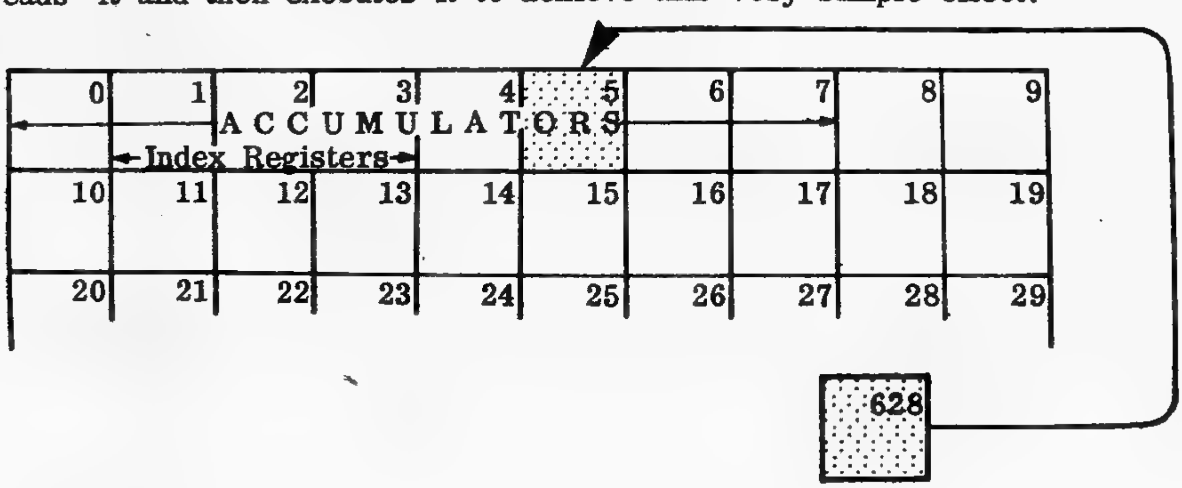
As you've seen, the computer word would hold the instruction like this:

5	000	not used	628

or, if you want to look at all the bits:

1 0 1	0 0 0 0 0 0 0	0 0	0 0 1 0 0 1 1 1 0 1 0 0

When the program is running and comes to this instruction the computer 'reads' it and then executes it to achieve this very simple effect:



Now let's go mad and use M.

We'll stick to the same basic instruction, but use M to call upon Index Register 3 to add its contents to the ADDRESS 628.

Here we are then:

F	X	N	(M)
000	5	628	3

held in a 1900 core storage word as:

5	000	3	628

or, if we show all the bits again:

1 0 1	0 0 0 0 0 0 0	1 1	0 0 1 0 0 1 1 1 0 1 0 0

Now look at Index Register 3.

0	1	2	3	4
000000000000000000000000	000000000000000000000000	000000000000000000000000	000000000000000000000111	00000000

INDEX REGISTERS

Automatically, on analysing the instruction the computer will add the contents of Index Register 3 to the address at N.

Write down the address of the core storage word which will have its contents loaded into Accumulator 5.

635 is the correct answer.

Index Register 3 contained seven— $\boxed{0 \text{———} 000111}$

This value was automatically added to the address at N, 628.

And, 628 + 7 = 635. So, the core storage word addressed by the instruction was number 635—**The basic instruction remains unchanged.**

Why did we not address 635 quite simply in the first instance and again ignore the business of address modification?

This question may be a bit of a tall order for you. Here's a clue. Remember that the computer rarely works through a list of program instructions once only.

In a business application, thousands of records may take their turn and be processed each by a PASS through the program. Even when this is not so—when only one pass of the program is required, say, to work some giant formula—certain series of instructions will form **loops** to gain repetitive effects.

If you've not already done so, make sure you have a good shot at the question before you go on.

The answer is that the programmer can arrange for the contents of Index Register 3 to vary, say, from pass to pass according to the type of record. Of course, this will require earlier instructions to load Index Register 3 appropriately.

It might be loaded from the input record with a factor which may change from record to record.

You saw that when Index Register 3 contained 7, word 635 was required in Accumulator 5.

At what core storage address would the programmer store the data required for loading into Accumulator 5 for an input factor of 10?

Go back over the last three pages again if you're not sure. Then write down the address of the core storage word before you go on.

The answer is 638. If you were wrong, follow the argument of the last three pages more carefully still.

We said earlier that the program's capacity for self-change is extraordinarily powerful.

Our earlier example of instruction modification was very simple, and the one you've just seen is also elementary.

Don't let that discourage you. If you've managed to stay with it through a first look at these basic ideas, then, computers need hold no terrors for you.

The expert programmer must follow the same steps. His expertise merely allows him to weave more elaborate patterns with similarly basic threads.

However, he uses a very helpful shorthand, and this will be our last topic before the end-of-section checkpoint.

You saw that the Function Code for the instruction Load Accumulator is 000. The Function Code for Add to Accumulator is 001; for Subtract from Accumulator, 003; for Binary to Decimal Conversion, 047.

The programmer is allowed to write his instructions in a more easily remembered code. The mnemonic (easily remembered) code for 000 is LDX; for 001, ADX; for 003, SBX; for 047, CBD.

This is called a **programming language**, and the I.C.T. 1900's programming language is called PLAN.

Addresses are also given easily remembered labels. The programmer chooses these for himself.

For example, if the programmer had chosen DATA as the basic address, the instruction you last studied, 000 5 628(3), would read:

```
LDX    5    DATA(3)
```

You might object that for the Function Code 000 the computer requires 0000000 to be stored in the F part of the instruction word, and how can the letters LDX be so interpreted? You know that these three characters would be represented by no less than eighteen binary bits.

You would be quite right. A program written in PLAN **can't** be loaded **directly** into Core Storage to store the appropriate instruction words. This is quite impossible. These alphabetical characters are stored as you've seen, in a BCD arrangement which is quite different from the Function Codes they represent.

But, since PLAN is fixed, the problem is permanent and unchanging. Why not have a permanent 'translation' program specially written to analyse the bit structure of mnemonics like LDX and substitute the appropriate seven bit code?

The special program is called an **ASSEMBLER**. It also allocates actual machine addresses to labels like DATA, checks the program for errors, and finally loads each instruction into Core Storage in the correct 24-bit form.

CHECKPOINT

Insert the appropriate words to complete the following statements.

1. The binary on/off device is called a ___.

2. A group of character locations or pure binary locations is called a _____.

3. A _____ _____-_____ machine has the same number of locations in each group.

4. Any number of locations may be grouped in a _____ _____-_____ machine.

5. So that the computer may be instructed to work on the appropriate data factor in Core Storage the data factor must be _______.

6. Core Storage holds both the data and the _______.

7. Instructions are held in a set Instruction _______.

8. A computer alternates continuously between _______ phase and _______ phase.

9. Instructions can be processed just like data. This is called _________ _________.

10. ________ are special words in which arithmetic results are held.

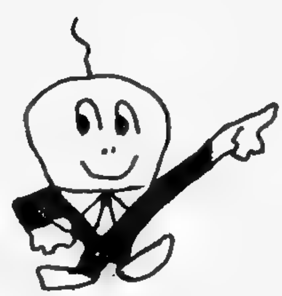
11. Index Registers may be used to effect a more automatic form of _________ _________.

12. An _______ translates a programming language into machine code.

CHECKPOINT ANSWERS

1. Bit
2. Word
3. Fixed Word-length
4. Variable Word-length
5. Addressable or Addressed
6. Program
7. Format
8. Instruction Phase Execution Phase
9. Instruction Modification
10. Accumulators
11. Instruction Modification
12. Assembler

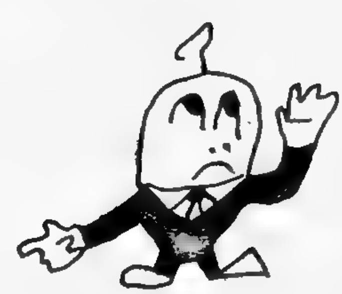
If you scored 10 or more,



You're ready to start the next section

If you scored less than 10,

You need to look at this last section again



SECTION IV

BASIC COMPUTER PROGRAMMING TECHNIQUES

At the end of Section III you saw that the computer programmer uses a programming language—PLAN for the I.C.T. 1900—and this frees him from the tedious business of keeping track of numeric addresses and gives him a more easily remembered version of the operation codes.

However, at the learning stage, programming languages present us with a problem. You've seen that instructions are made up of actual characters which the computer holds in Core Storage like data, and in meeting the basic programming techniques this is a firmer, more realistic base than the 'once-removed' programming language.

So, as you meet the basic programming techniques in the earlier part of this section, we try to keep you very close to the idea of instructions stored in Core Storage.

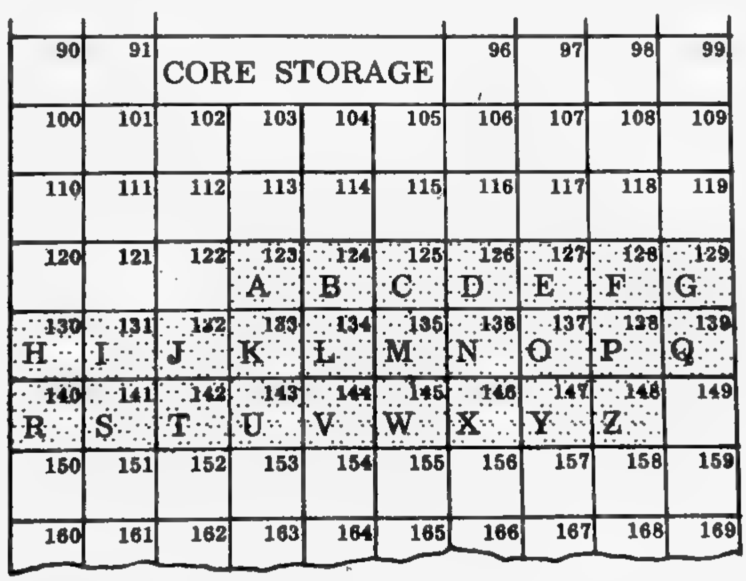
Later in the section you'll see that, in flow-charting and in writing programming language on a Coding sheet, the basic programming techniques still apply.

The simplest of programs is an invariable series of instructions stored in the sequence in which it must be performed.

The following rule applies to nearly all digital computers:

UNLESS YOU INSERT AN INSTRUCTION TO THE CONTRARY THE COMPUTER WILL ALWAYS OBEY THE NEXT SEQUENTIALLY STORED INSTRUCTION.

Let's look at a simple diagram of a straight sequential program in Core Storage:



The computer obeys instructions A, B, C, D and so on to Z, **in that order**.

None of the instructions is a BRANCH (jump) instruction. Thus, the computer obeys the twenty-six instructions starting at A and finishing with Z.

Do you think the computer will **automatically begin again** at instruction A to perform the same program on the next input record?

Write down your answer, yes or no.

The answer is 'No'.

We said that none of the instructions is the sort which over-rides the 'next sequential instruction' rule. After Z, the computer will expect to find the next instruction stored in WORD ____ ?

Refer to the diagram again, and write down the word number.

The answer: 149.

So, the diagram shows a program which the computer will work through **automatically** once only.

The computer OPERATOR tells the machine—he can do this externally—where to begin.

In our simple example he would need to do this for every pass of the program.

The operator would 'key-in' the address of the first instruction—123 in our example.

And, because we've chosen the simplest possible example, the computer would obey the instructions in sequence and finally expect to find an instruction in WORD 149.

Now consider this extension to the program:

You wish to extend your program by adding on four instructions at the end.

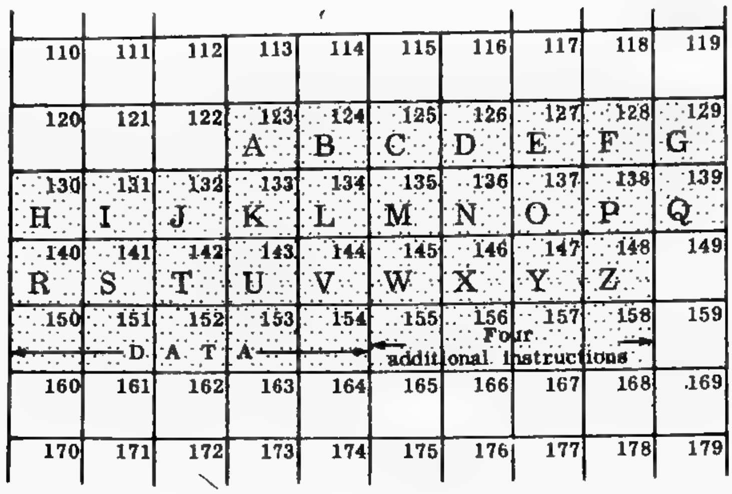
Unfortunately, you happen to know that the program uses WORDS 150, 151 152, 153 and 154 to hold data:

CORE STORAGE

110	111	112	113	114	115	116	117	118	119
120	121	122	123 A	124 B	125 C	126 D	127 E	128 F	129 G
130 H	131 I	132 J	133 K	134 L	135 M	136 N	137 O	138 P	139 Q
140 R	141 S	142 T	143 U	144 V	145 W	146 X	147 Y	148 Z	149
150 ←	151 D	152 A T	153 A	154 →	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169
170	171	172	173	174	175	176	177	178	179

However, you're not completely out of luck: WORD 149 is available, and so are all the WORDS from 155 onwards.

This is the situation, then:



We've stored the four additional instructions at 155, 156, 157 and 158.

After Z, the computer expects to find the next instruction at 149. We want the program to branch (jump) to 155.

At 149 we store a BRANCH INSTRUCTION which contains the address, 155. In obeying this Branch instruction at 149, the computer ignores the address at which it would normally seek the next instruction (150) and jumps instead to the address in the branch instruction:

Branch Instruction at 149 orders: 'Don't go to 150 for the next instruction as you normally would. Go to 155 instead'.

Would you say that this sort of Branch Instruction depends on a certain condition or would you call it unconditional?

No condition is involved, and we are using the **UNCONDITIONAL BRANCH** Instruction.

When the computer reaches 149 it is instructed to branch to 155. There's no question of 'If . . .' or 'Unless . . .': we want the program to jump to those four additional instructions, **unconditionally**.

The Unconditional Branch instruction is very simple. Do you remember the format of I.C.T. 1900 instructions?

F Operation Code	X Accumulator	N The Address of a Core Storage Word	(M) Modifying Index Register

The 1900 mnemonic operation code for the Unconditional Branch instruction is BRN.

Use the boxes provided above and write in the Unconditional Branch instruction which we decided to store at 149.

ANSWER:

F Operation Code	X Accumulator	N The Address of a Core Storage Word	(M) Modifying Index Register
BRN		155	

If you answer was incorrect, read the last three pages again.

This Unconditional Branch instruction must be stored in WORD 149.

You may be wondering how the programmer arranges for this—for instructions to be stored in the appropriate words.

In more primitive computer times, the programmer's list of instructions, **each with the Core Storage address at which it was to be stored**, would be punched into cards. A special program loading routine would then store the instructions, each at its specified address.

Everything was completely straightforward. As he wrote his list of instructions on a Coding sheet, the programmer recorded alongside each instruction the address at which it was to be stored. He worked on this real address basis throughout.

Nowadays, the programmer doesn't concern himself with actual machine addresses. He says, in effect, 'I'll store such and such instruction at LOOP. Then, when I want the program to jump back to this instruction, I'll write an instruction to branch to LOOP. I have a powerful translation program which will substitute the same real machine address whenever it encounters the symbolic address, LOOP'.

Let's use our example to illustrate both methods of writing instructions on the programmer's CODING SHEET.

We'll make no distinction between instructions in machine code and in programming language except for the Unconditional Branch.

Our point is the distinction between actual machine addresses **of instructions** and relative addresses.

MACHINE CODE
(actual addresses)

ADDRESS OF INSTRUCTION	INSTRUCTION
123	A
124	B
125	C
126	D
127	E
128	F
146	X
147	Y
148	Z
149	074 155
155	P
156	C
157	M
158	J

PROGRAMMING LANGUAGE
(symbolic or relative addresses)

ADDRESS OF INSTRUCTION	INSTRUCTION
BEGIN	A
	B
	C
	D
	E
	F
	X
	Y
	Z
	BRN EXTRA
EXTRA	P
	C
	M
	J

Why do you think the symbolic program omits any sort of address for so many instructions?

Write down your answer.

The answer, of course, is that they're not needed.

When it encounters a symbolic address for an instruction, the special ASSEMBLER program allocates an actual machine address.

Whenever it encounters the same symbolic address WITHIN AN INSTRUCTION, it substitutes the same actual address.

Where it encounters no **address for an instruction** it knows that it **MUST** store the instruction in the **next sequential** Core Storage word.

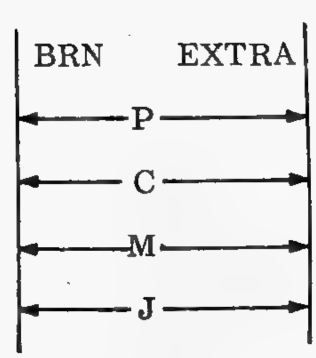
So, if you answered to the effect that those instructions with no symbolic **INSTRUCTION** address must slearly be sequentially stored, your answer was perfectly adequate.

If you're not satisfied with your answer, take another look at the two sorts of coding sheet on the last page.

Now try this question:

Must these additional instructions be **written** on the Coding Sheet immediately after the BRN EXTRA line?

EXTRA

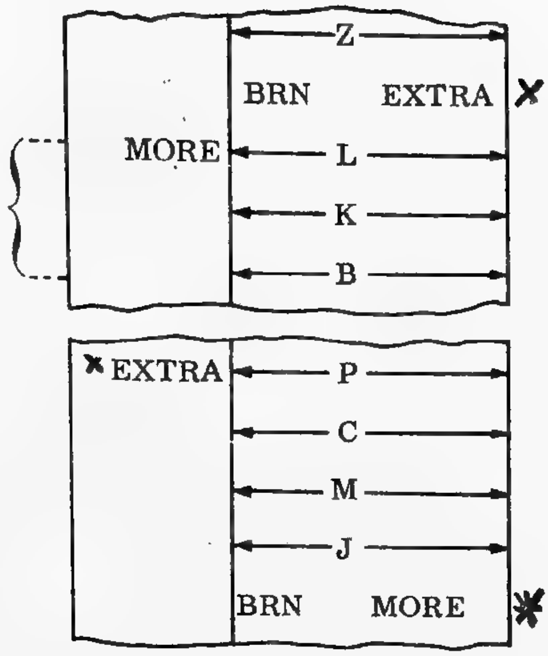


Write down your answer, 'Yes' or 'No'.

The correct answer: 'No'.

This is why:

A further series of additional instructions



Just as data might break the sequence of instructions in storage, so might yet another series of instructions.

Remember, our example is still the simple 'straight-line' program, but, as you see, you can jump about a bit in writing it down, and consequently in storing it.

The program's straight sequence begins:

A———Z, BRN EXTRA (to jump those data words in 150 to 154),.....

Write down the rest of the straight sequence of instructions **as they will be performed** now that we've added the two lots of additional instructions.

The correct answer: P, C, M, J, BRN (to jump to MORE), L, K, B.

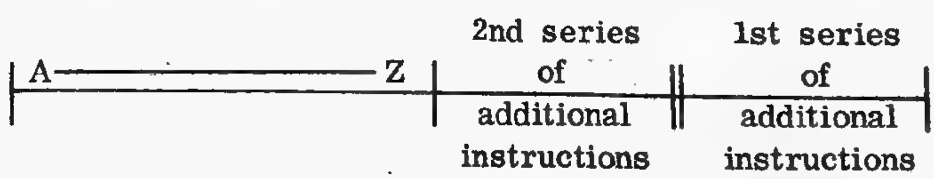
If you were wrong, trace through the coding on the previous page again.

Remember, out term 'next sequential sequence' refers to the next **sequentially stored** instruction.

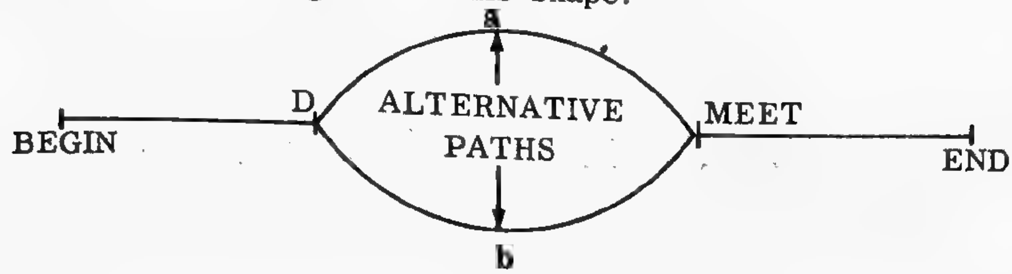
You've now seen that a series of instructions can be **performed** in straight sequence although they may be stored in several blocks.

The Unconditional Branch instruction allows the program to jump unconditionally from block to block.

The program you've just seen had three blocks of instruction in this shape:



Now let's consider a program of this shape:



Here, the computer must make a decision.

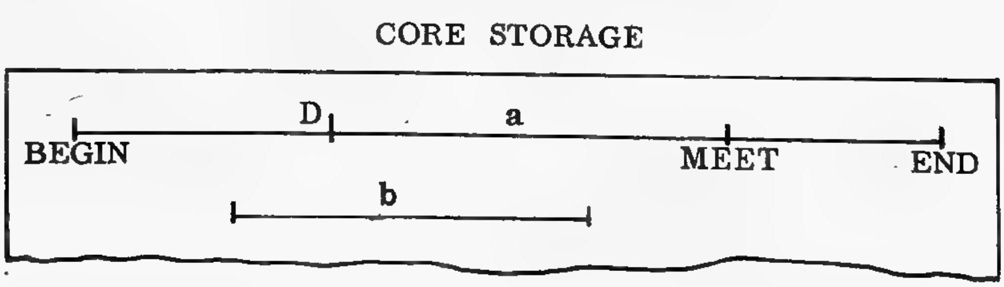
Let's suppose that the result of a calculation early in the program is tested at decision point, D.

If the answer is negative, one alternative series of instructions, **a**, is obeyed. If the answer is not negative, i.e. if it is zero or positive, **b** is obeyed.

How would this program be stored in Core Storage?

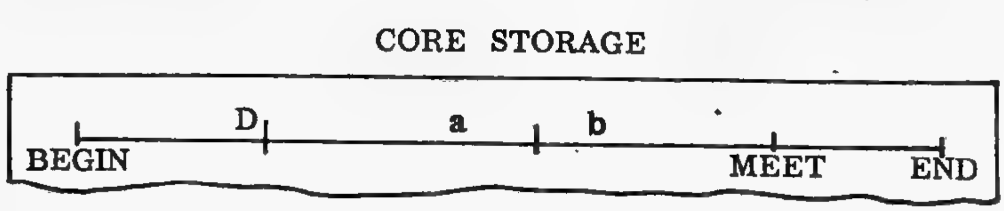
Like this? –

(i)



Or like this? –

(ii)



(iii) Or, are both Core Storage arrangements, (i) and (ii) valid?

You've yet to meet a decision-making instruction, but you should be able to answer on the strength of what you know already.

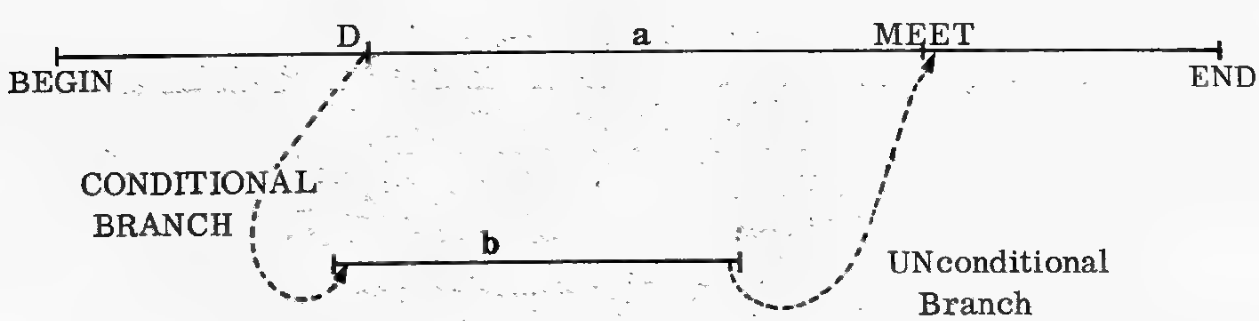
The correct answer is (iii). Both (i) and (ii) are equally good ways of storing this program.

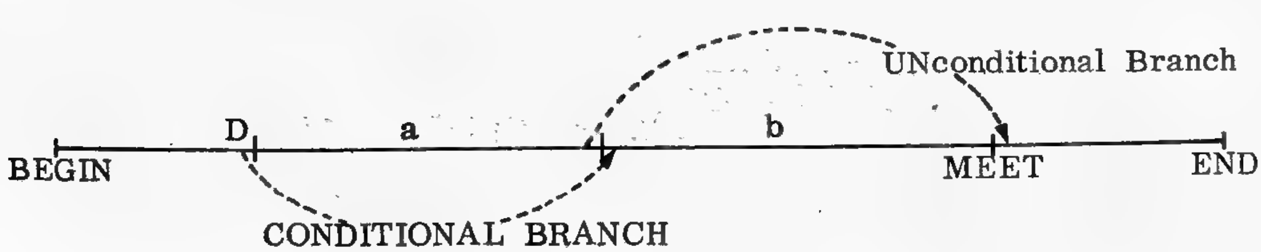
Clearly, only one series of instructions can follow the decision point sequentially.

The alternative series must be stored somewhere else—and it doesn't much matter where.

And, as no doubt you saw, we shall still need the UNCONDITIONAL BRANCH instruction to jump from one of the alternative blocks to MEET.

Look at the two versions again:





Work through **each** version **twice**, once taking the **a** path and once taking **b**.

CONDITIONAL BRANCH instructions enable the computer to make decisions.

A computer usually has several Conditional Branch instructions each capable of making a different sort of decision.

In our example we require the decision on the basis of a negative or non-negative result—in an Accumulator, in fact.

On the I.C.T. 1900 we'd choose the BPZ Conditional Branch instruction for our example—Branch on Positive or Zero.

We want to take path **a** on the negative condition. BPZ will branch the program to the **b** path on positive or zero.

For a result which does **not** meet the condition, positive or zero, i.e. on a negative result, the computer takes the next sequential instruction.

So the rule is:

> **BRANCH** if the condition is met,
>
> **next sequential instruction** if the condition is not met.

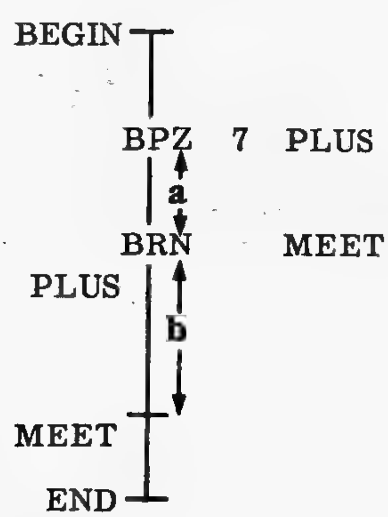
You can probably guess how the BPZ instruction is made up:

F	X	N	(M)
The Operation Code	Accumulator	Address of a Core Storage Word	Modifier
BPZ	7	PLUS	

7 is the Accumulator containing the result to be tested.

PLUS is the 'Jump-to' address—in our example the address of the first instruction in the **b** block.

Let's look at a rough outline of our program:



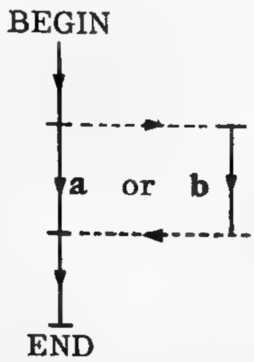
Insert the words UNCONDITIONALLY and CONDITIONALLY, in the following sentences:

The program jumps to PLUS __________.

If it takes the **a** path, it jumps to MEET __________.

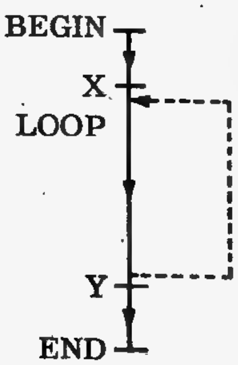
The program jumps to PLUS, CONDITIONALLY. If it takes the **a** path, it jumps to MEET, UNCONDITIONALLY.

So, you've now seen the most simple decision-making program—incorporating a choice between two alternative lines of action.



Now for a key question:

How would you make a program loop?

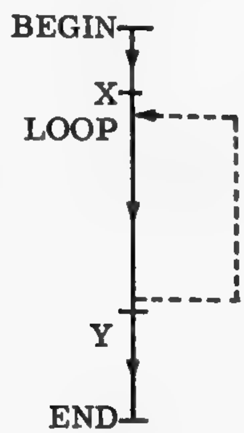


What would you do and where would you do it?

Write down your answer.

We hope you'd insert, immediately before Y, a CONDITIONAL BRANCH to LOOP.

If your answer was wrong, here's the sketch again:

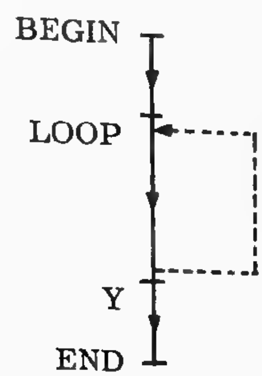


A loop has the effect of merely repeating a series of instructions.

We shall achieve this by branching from the end of the series back to the beginning.

Why did we not choose an UNCONDITIONAL BRANCH instruction?

Yes, of course. If you jump back with an **Unconditional** Branch instruction, the series of instructions beginning at LOOP will be repeated again and again and again ...... without stopping.



The decision, therefore, is between 'Go to LOOP' (i.e. jump), and 'Go to Y (the next sequential instruction)'. It must be a **CONDITIONAL** Branch or there would be no way out of the loop.

Let's suppose that a certain process demands six passes through the series of instructions which begin at LOOP.

To control the number of passes, we set up a **COUNT**.

You load the value 6 into an Accumulator as your COUNT, and subtract 1 from it every time you go through the series of instructions. When the series of instructions has been performed 6 times, your COUNT has been reduced to zero.

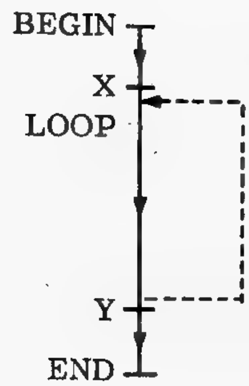
This basic programming technique merely requires a Conditional Branch instruction which tests for zero.

The I.C.T. 1900 has a particularly powerful instruction for this purpose. It not only tests for zero, but first subtracts the 1 from the COUNT as well! (These are only two of its accomplishments: we need not concern ourselves with its full potential.)

We set up a COUNT of 6, then. Until it reaches zero we want the program to loop.

Should this Conditional Branch instruction **branch** on zero or non-zero?

Answer: On non-zero.



We are decrementing a count of 6 to control six passes of the instructions which start at LOOP.

The Conditional Branch instruction **tests** for zero, and must **branch** on **non-zero.** When the count is reduced to zero we want to continue sequentially at Y.

On the I.C.T. 1900 we'd use the BUX Conditional Branch instruction.

This instruction will decrement the COUNT in the ACCUMULATOR specified in the instruction, and branch the program to the Core Storage address specified in the instruction—until the count is reduced to zero.

BUX 3 LOOP

Where in our short program, would you have loaded Accumulator 3 with the count, 6?

Answer: At the beginning, before LOOP. (Clearly, if the count were renewed as the program looped, you'd defeat your object—and the loop would go on and on and on . . . .)

Let's summarise what you've seen so far in this section:

1. Even the simplest once-only program may need to branch to a non-sequential instruction address. You use an UNCONDITIONAL BRANCH INSTRUCTION.

2. When your program decides between two alternative series of instructions, only one series can be stored sequentially. You branch to the alternative series CONDITIONALLY with a CONDITIONAL BRANCH INSTRUCTION.

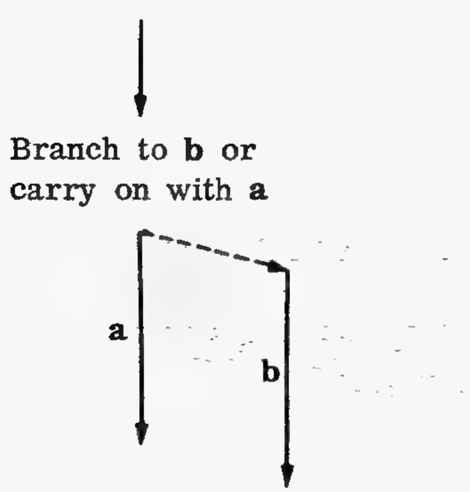
3. Computers offer a range of CONDITIONAL BRANCH INSTRUCTIONS to meet a range of decision-making conditions.

4. LOOPING, often controlled by a COUNT, is achieved by using a CONDITIONAL BRANCH INSTRUCTION which also allows escape from the loop when the branching condition is no longer met.

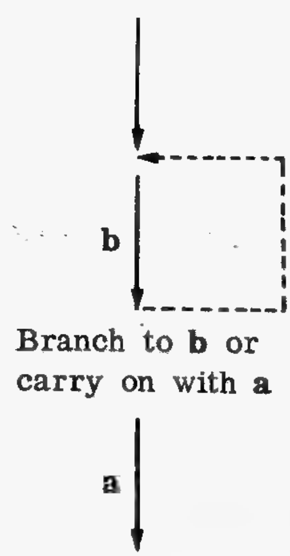
These techniques, then, are the basic elements in any program:

> **Unconditional** branching when the next instruction to be performed is **not** the next in Core Storage.
>
> Testing for a condition and **conditionally** branching to one of two alternative routines.
>
> **Looping** to repeat a series of instructions (or the whole, or a large part of, the program).
>
> Controlling a loop with a **count** and a conditional branch instruction.

Strictly speaking, a loop is no more than any other conditional branch:



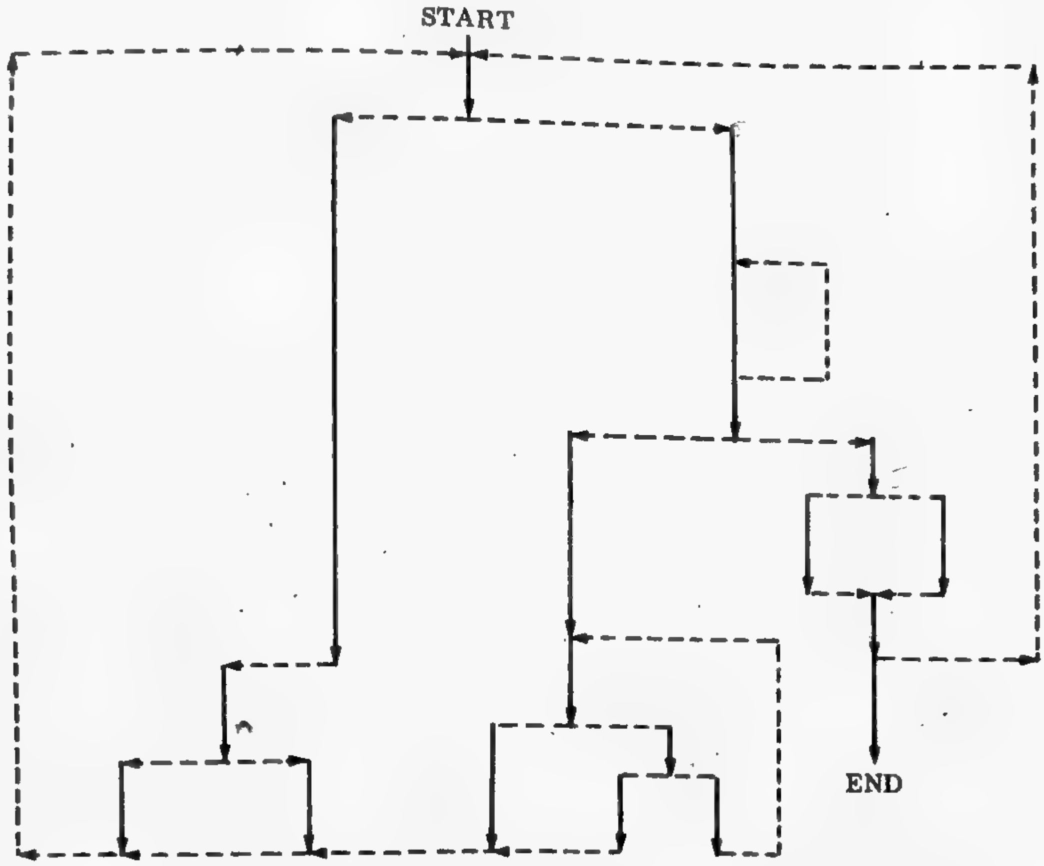
A conditional branch
an alternative routine



A loop

Now let's look at a complete flow with several branches and loops.

The basic techniques of computer programming are few. A complicated program is made up of many branches and loops.



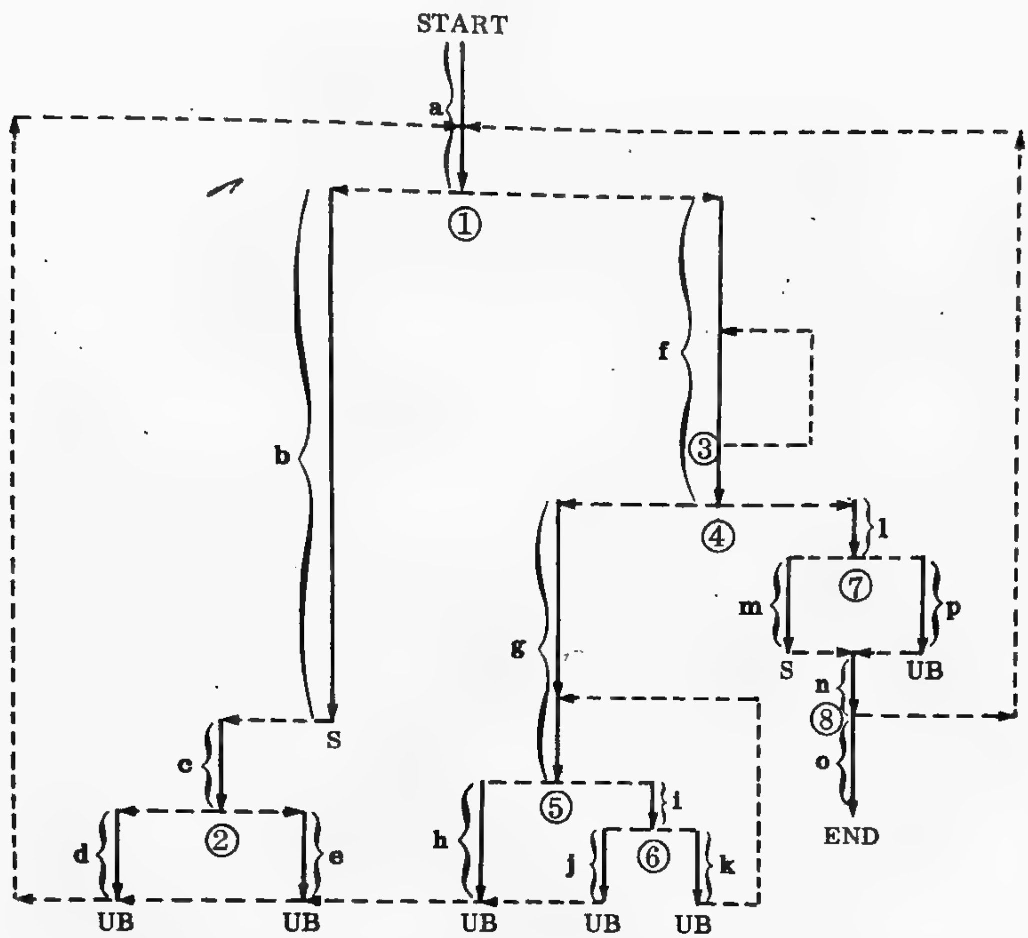
The solid lines represent the series of instructions. The broken lines fill in the possible courses the program may take.

N.B. This is a simplified representation of a program flow, and **not** the full form of program flow-chart which the programmer uses. We'll come presently to the actual flow-charting conventions.

How many Conditional Branch instructions will be used in the program represented above?

Answer: 8.

Here they are, marked ① to ⑧ :



The diagram also shows where **Unconditional** Branch instructions will be necessary—marked UB, and two points marked S, to show that the next series of instructions must be sequentially stored.

Remember too, that at the **Conditional** Branch points, one of the two alternative series of instructions must follow sequentially in Core Storage.

Now, for the sake of uniformity, at Conditional Branches regard the **left-hand** alternative as the sequential series, and the **right-hand** alternative as the 'branched-to' series.

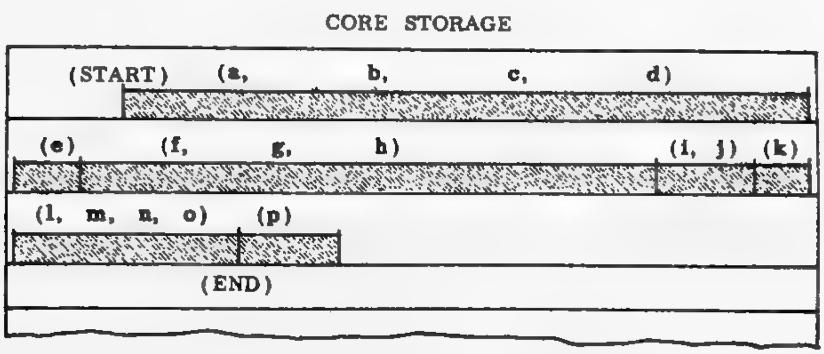
Write down the series **a, b, c,** etc. **in the order in which the programmer would write them** on a coding sheet ready to be stored in Core Storage.

Answer: **(a, b, c, d) (e) (f, g, h) (i, j) (k) (l, m, n, o) (p).**

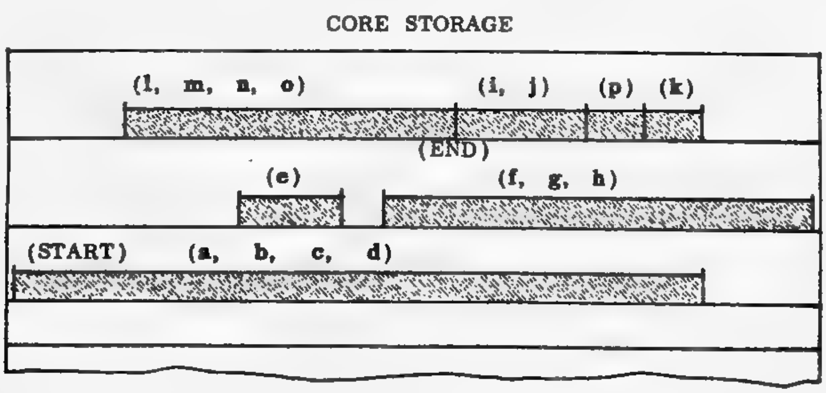
The sequences **within** the brackets are obligatory, but if you look at the program and think carefully, you'll see that it doesn't matter how you shuffle these groups—although, of course, there's no point in writing and storing them other than in the most straightforward sequence.

The beginning of each bracketed group is a 'known' address, i.e. the program can branch to it, and each group ends with an Unconditional Branch (or END in the case of **l, m, n, o**).

So, in Core Storage, your program might look like this:



or like this:



Any arrangement of these groups is possible.

The various Conditional Branch instructions will determine the particular course taken on any given pass through the program.

Following the flow-chart, the programmer simply writes the program instructions on the Coding Sheet. Where instructions must follow sequentially, he must write them sequentially, but the blocks of sequential instructions may be written—and thus **stored**—in any order.

It's perfectly easy to see from the flow-chart whether or not this or that order of writing will work.

So, you've seen that program instructions can be quite easily stored in Core Storage to allow both sequential performance and 'jumping'.

Do you remember what we said in Section III about the unlimited possibilities for instruction modification?

Because the computer stores the program instructions just like data, the programmer is free to write instructions which will, on the appropriate conditions, change other instructions. The programmer can take this principle to any lengths. Only the limits of his own ingenuity—and perhaps a salutory desire to keep his program reasonably straightforward—need hold him back.

Similarly, in this section you've seen how the programmer can write programs to solve data processing problems which involved **decision-making.** This too can be taken to any lengths.

And, like instruction modification, this vast facility is made possible by very simple principles:

1. The principle of 'left-to-right' sequential performance of instructions sequentially stored in Core Storage.
2. The principle of breaking this sequence with Branch instructions which jump the program to a specified address either unconditionally or on recognising a condition.

So far you've seen what this means in terms of instructions actually stored in Core Storage.

Now let's see how the programmer prepares his program.

In planning his programs, the programmer follows certain diagramming conventions which help him keep track of what he's doing.

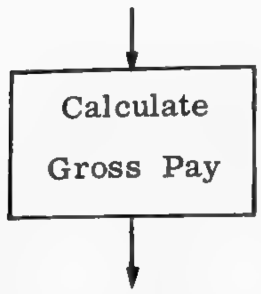
Briefly, these conventions provide a system of boxes joined together with arrows—a program FLOW-CHART.

So, the programmer designs his program as a FLOW-CHART which he subsequently follows as he writes the actual instructions in programming language on a CODING SHEET.

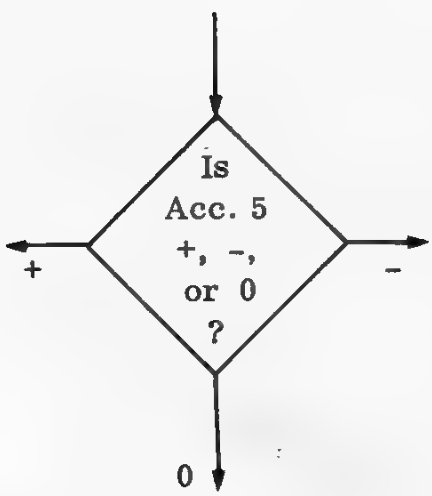
(First of all, of course, he must provide himself, or be provided with specifications which fully describe the job, but we are not concerned here with the entire procedure.)

Let's look now at the programmer's flow-charting symbols.

Here are the two most used symbols:



This ordinary rectangular box is used to contain a description of computing operations, e.g. Arithmetic operations, moving data, loading accumulators, etc. A whole series of such operations may be described in one box so long as they are required in straight sequence.



This is the decision box. A good shape because an arrow can lead into one corner and three alternative courses can come from the others. The required alternatives **may** depend upon more than one Conditional Branch instruction, e.g. testing for +, -, or 0. The box can show the three paths, but Conditional Branch instructions can usually decide between two courses only. Thus, two instructions would be necessary.

A program flow-chart box may represent more than one program instruction.

The programmer uses these two main boxes more than any others.

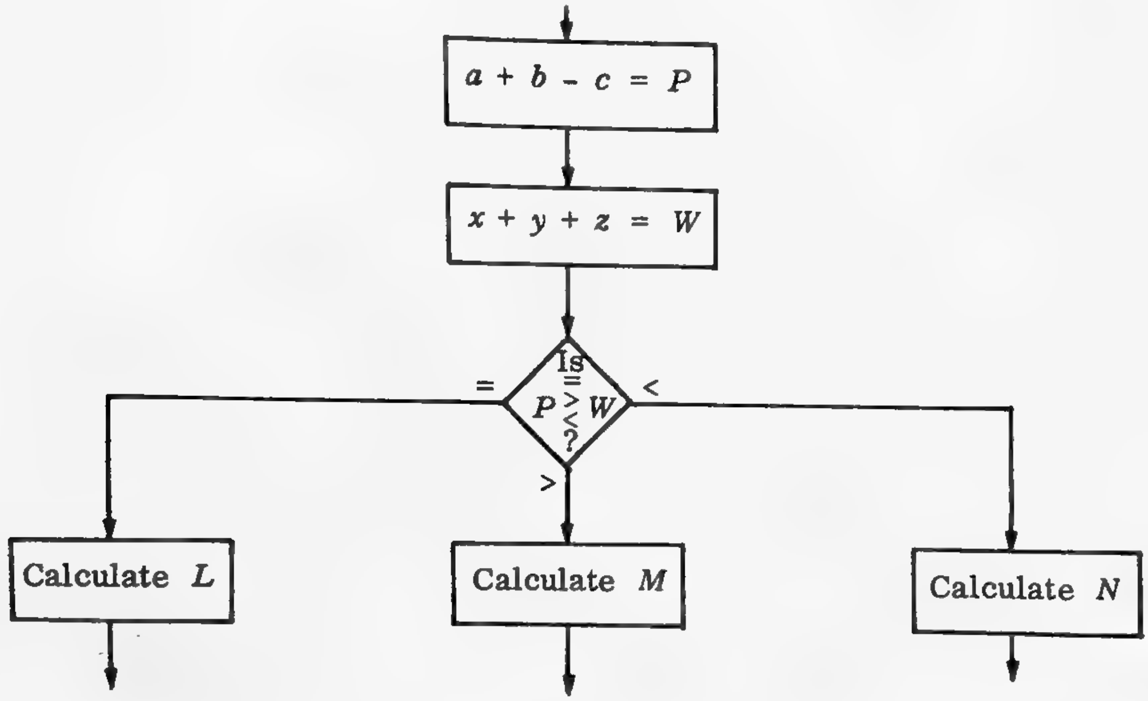
Let's take a simple example:

The job:- Calculate $a + b - c$ $(= P)$ and $x + y + z$ $(= W)$.

If $P = W$, calculate L.

If $P >$ (is greater than) W, calculate M.

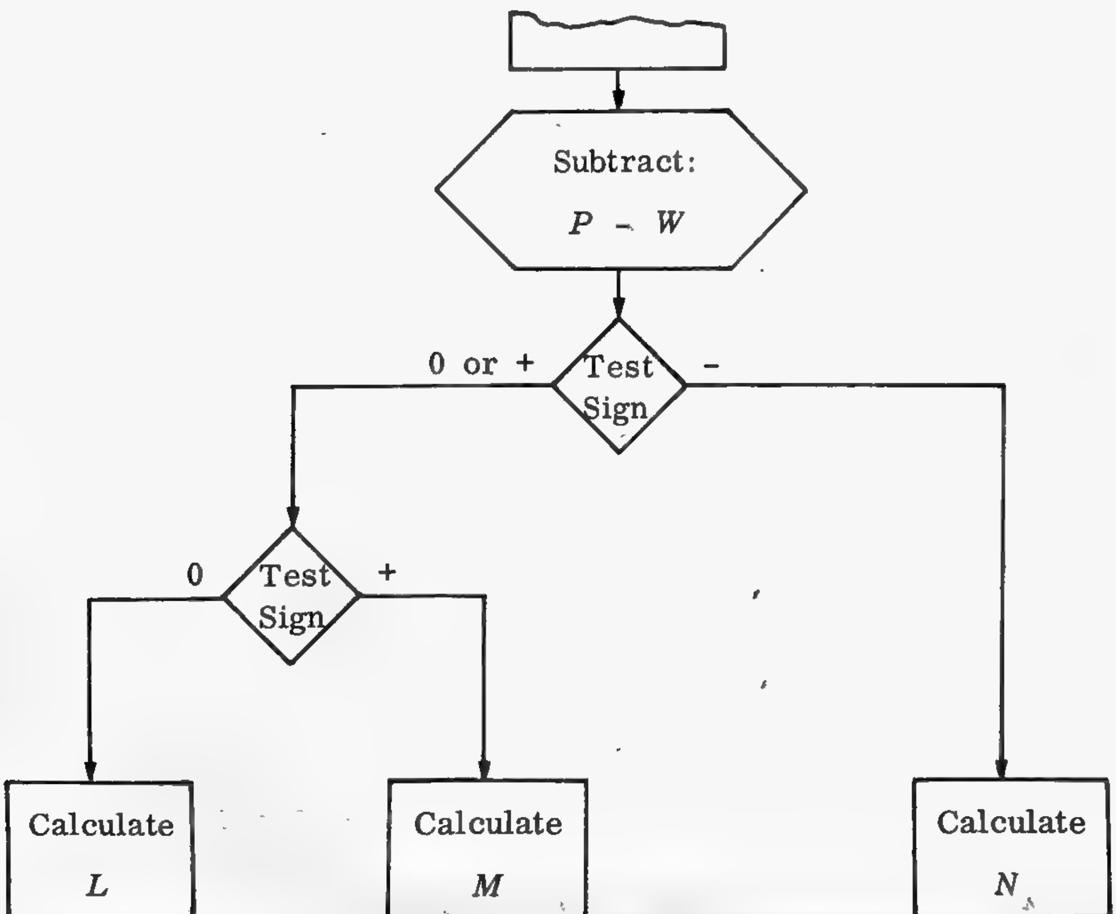
If $P <$ (is less than) W, calculate N.



Now, this is an abbreviation of what must be written eventually as program instructions, but it must be detailed enough to satisfy the programmer that, working with the available instructions, the job can be done in this way.

The programmer, then, must represent in his flow-chart ways and means which are compatible with the instructions available on the specific computer.

In our example he might, of course, have shown how W will be subtracted from P, and the full detail of the branching:



But, the programmer would not normally show the decision point in such detail. He **knows** that the briefer version implies **two** Conditional Branch instructions. He knows too that the three alternative courses emerge irrespective of more or less detail in the flow-chart.

Moreover, anyone else who is reasonably familiar with the specific computer's instructions can also work from the briefer version.

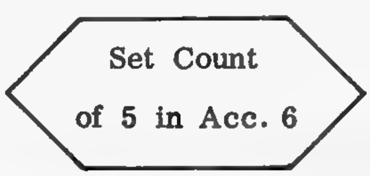
You no doubt noticed that we slipped in a new box on the previous page.

Because in this more detailed flow-chart we showed the actual **preparation** for the sign test we wrote that operation, $P - W$, in the Preparation box.

This box is used for an operation which expressly establishes a condition subsequently tested by a Conditional Branch instruction.

In the briefer version of our example, the programmer omitted any preparation for the test—because his intentions were obvious to him, and to anyone else in a position to use his flow-chart.

He **would** use this box to set up a COUNT:



And, he'd use it to set and unset a switch—a technique we'll now discuss.

The programmer sets a SWITCH—sometimes called a PROGRAM SWITCH to distinguish it from the hardware variety—by placing a character or a bit in a specified Core Storage position. Subsequent instructions can then test for the presence or absence of the switch.

A switch 'remembers' a condition.

You've seen how program instructions can recognise a condition, and **immediately** determine which of several courses the program must take, —like this:

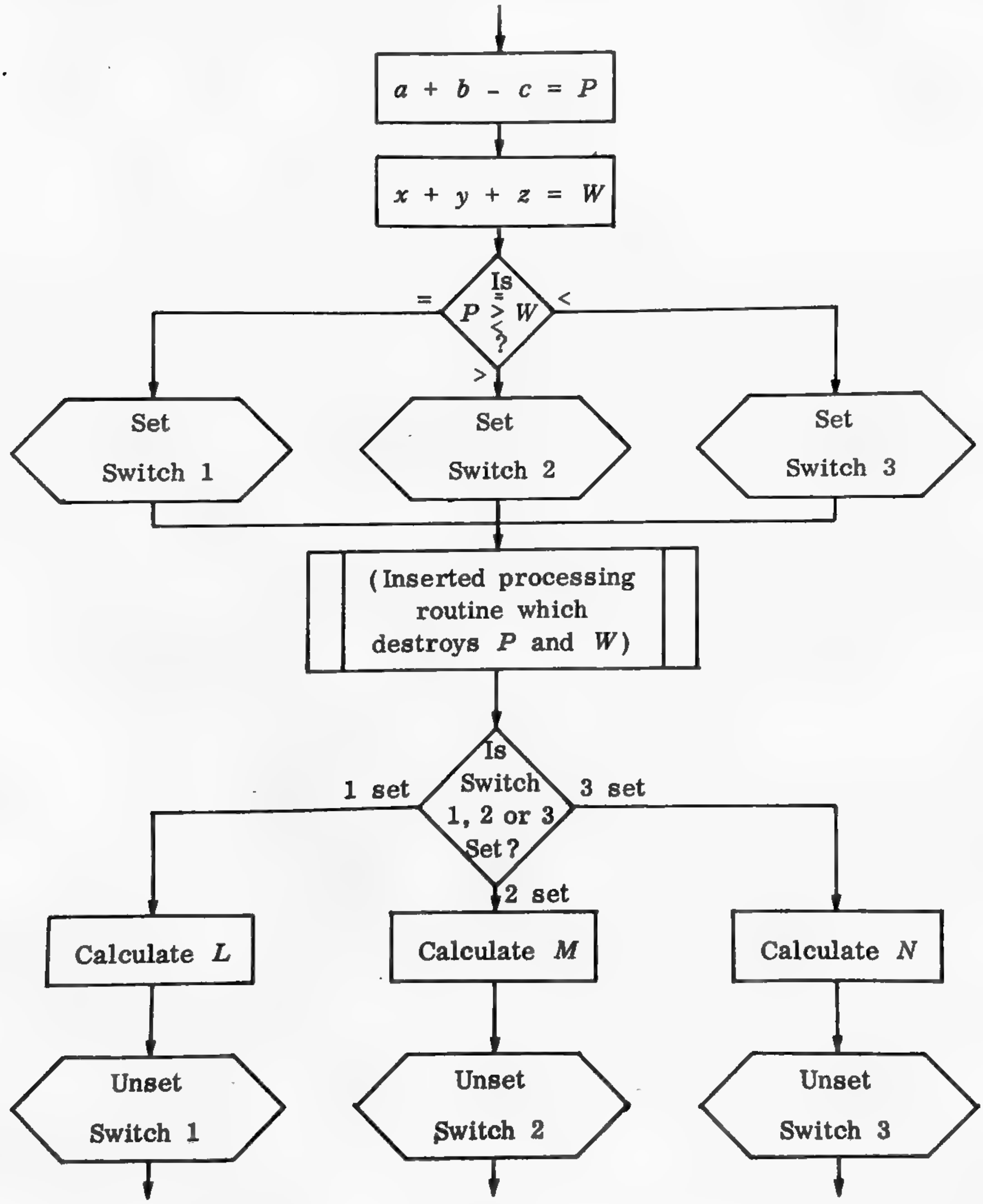
1. CONDITION TESTED
2. BRANCH ACCORDINGLY.

But what if the job demands that some other processing must intervene; and what if this other processing destroys the condition?

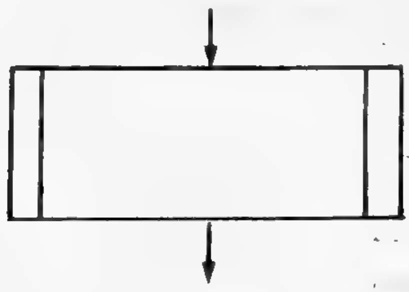
1. CONDITION TESTED (say, +, 0, or - in Acc. 6)
2. OTHER PROCESSING IN ACCUMULATOR 6
3. BRANCH ACCORDING TO 1.

Clearly, the original condition of Accumulator 6 must be 'remembered'. The programmer uses switches which continue to represent the original condition after it has been destroyed.

Suppose that, to develop our last example, several processes must precede the calculation of L, M, or N. Suppose, too, that this inserted processing destroys P and W, and some of their factors.



Again, we slipped in a new box on the last diagram:



This is the sub-routine box.

A sub-routine is a standard, pre-programmed series of instructions for a commonly used bit of program. It saves the programmer the unnecessary fag of writing similar routines in many programs.

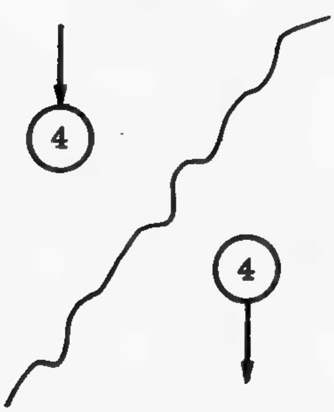
Thus a standard sub-routine serves many programs and many programmers.

Usually, a single instruction in the programming language program calls upon the ASSEMBLER program to include the sub-routine in the machine language program.

Let's quickly introduce the remaining boxes.

This is the Input/Output box. It's used for any operation which transfers data to or from a file, display, enquiry or command station.

This is the Terminal box used to mark the beginning and the end of the run. It's also used to mark programmed halts which require operator attention.

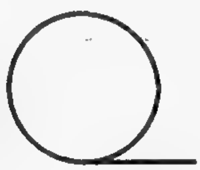


A small circle is used to signify, 'Continue from small circle with same legend'. This device leads the flow from one page to the next—or perhaps from an awkward corner on a large sheet to a better continuation point.

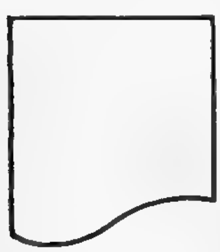
These symbols are the subject of international standards agreements, but you may come across flow-charts which use different symbols, or which use the same symbols differently.

A common practice in the past was to use specific symbols for input/output functions, like those on a SYSTEMS flow-chart:

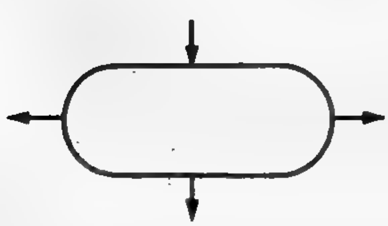
e.g.



Some programmers may use this symbol, or something like it, for magnetic-tape input and output . . . .



. . . . and this for line printer output.



The box with rounded ends—now used for START, END and HALT—has been used in the past as the decision box.

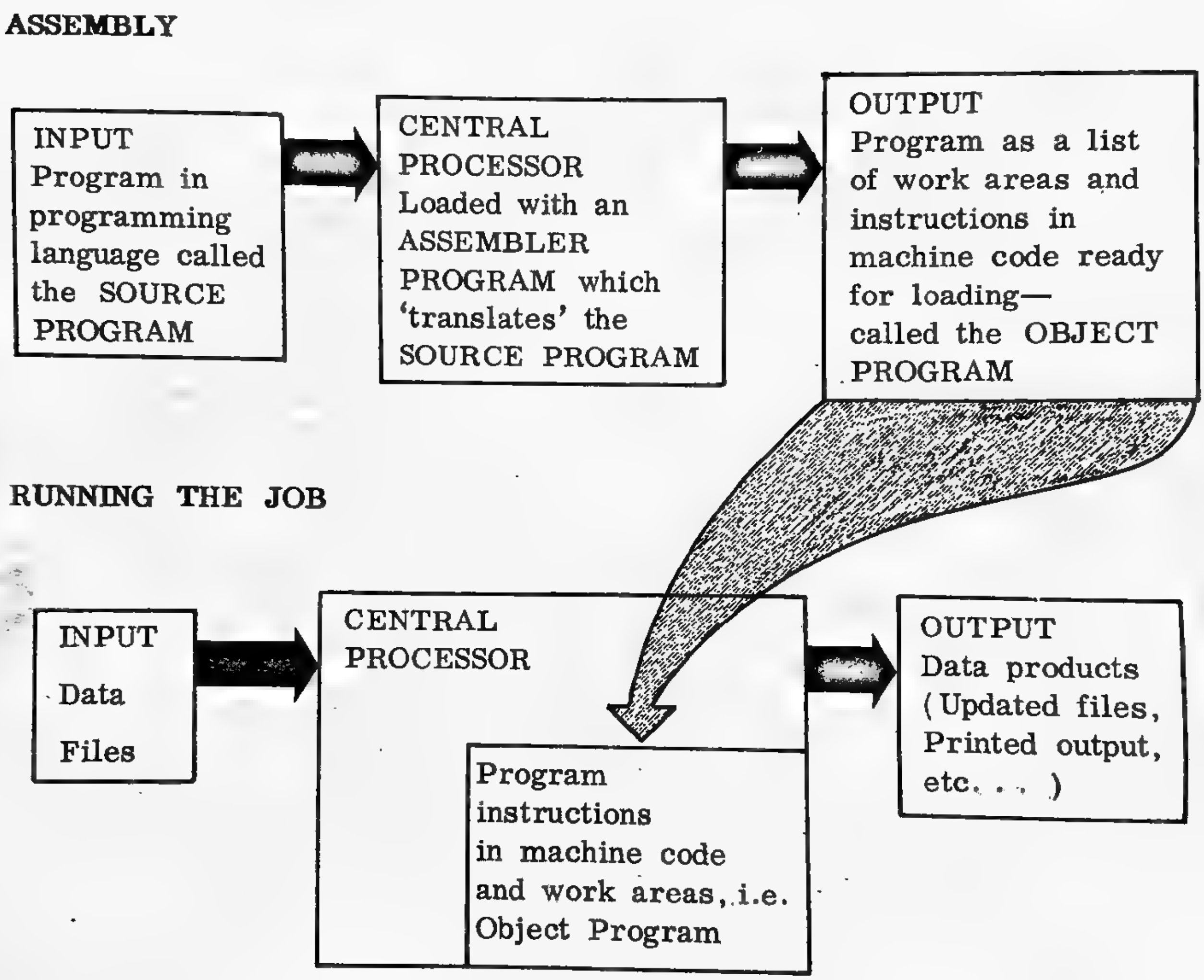
However, both past and present variations are pretty well self-explanatory.

The programmer draws the symbols long and thin or short and fat to suit his purpose. Size and proportion are not standard.

Finally in this section you'll see a very short and simple but complete program flow-chart followed by the actual programming language lines which are coded from it.

Before we go on to this, let's be clear about the program instructions as the computer obeys them in doing the job, and the programming language lines as written by the programmer.

Consider this diagram:



Notice that we've introduced two new terms. What the programmer writes in programming language is called the **SOURCE PROGRAM**.

This must be 'translated' by an ASSEMBLER program into the machine language program which the computer obeys—the **OBJECT PROGRAM**.

'SOURCE PROGRAM', then, is a clear and easy way of saying, 'the program written in programming language by the programmer'.

Similarly, 'OBJECT PROGRAM' is a good way of saying, 'the program as it finally stands with the instructions and addresses in actual machine code ready to be loaded into the processor'.

Although it saves the considerable tedium of keeping track of machine addresses, you must not expect the Source Program to be quite as straightforward as a simple list of instructions in machine code.

In the old days, programmers wrote the machine code object program straight away. They could see from their Core Storage chart the exact machine code address for everything they intended to store—each data factor and each instruction.

Nowadays, you leave all this to the Assembly process. But this means that your Source Program must include not only your instructions, but also certain **DIRECTIVES** and **DATA STATEMENTS** which state the program's storage and addressing requirements.

You'll see these presently in our coding.

Here's our very simple, complete job:

CARD REPRODUCING

* With the exception of a certain class of card, a file of punched cards must be reproduced, i.e. new cards must be punched as exact copies of the originals.

* Cards with a '9' hole punched in card column 1 must **not** be reproduced. (**All** the cards are punched in card column 1 with various punched holes from '1' to '9'.)

* To mark the end of the card file, a blank card is placed there. It can therefore be distinguished as the end of file marker because it has no value punched in card column 1.

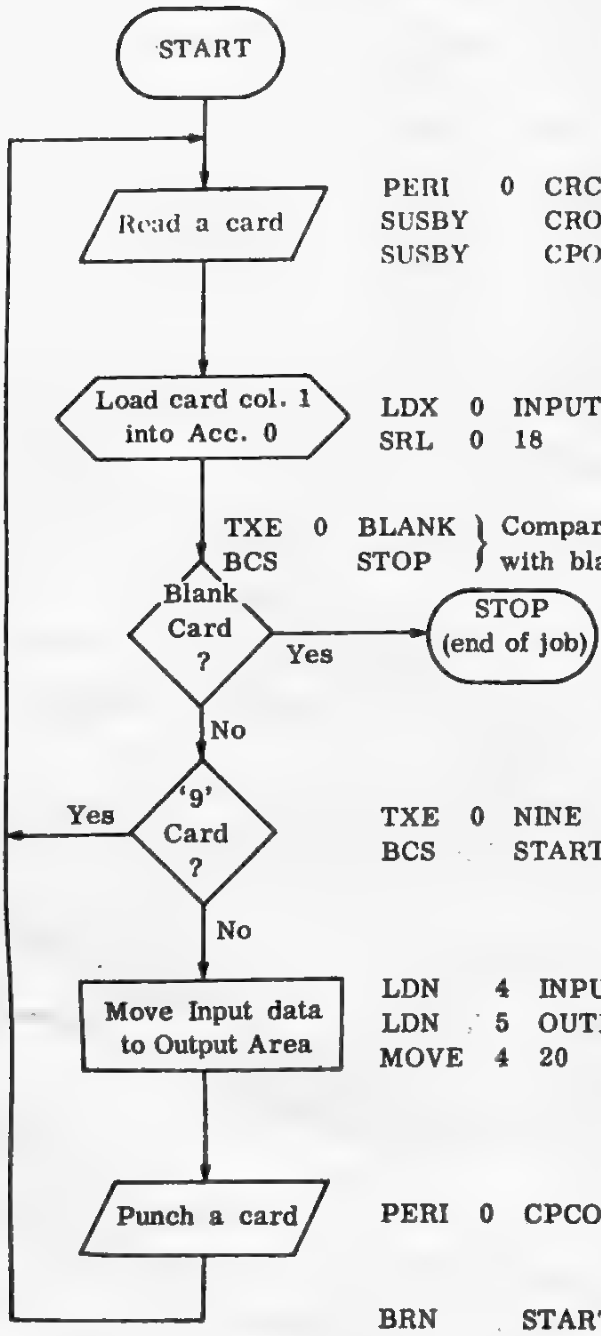
(Please note that this job has been chosen not because it's a typical computer task—it isn't—but because it makes a brief and simple example.)

The program flow-chart appears on the next page, and alongside each box we've shown the **INSTRUCTION** lines as they appear on the coding sheet which follows.

All other coding sheet lines are either DIRECTIVES or DATA STATEMENTS which inform the ASSEMBLER program to reserve and address data areas, for example.

The program is written in PLAN for the I.C.T. 1900.

PROGRAM FLOW-CHART—CARD REPRODUCING



PERI	0	CRCON	Peripheral instruction to read a card followed by suspension of the program until the card read and punch operations are complete
SUSBY		CRO	
SUSBY		CPO	
LDX	0	INPUT	Card column 1 character loaded into, and shifted down to the junior end of ACCUMULATOR 0
SRL	0	18	
TXE	0	BLANK	Compare contents of Acc. 0 with blank and branch if equal
BCS		STOP	
SUSWT		CR	Stop and type out 'Halted CR'
TXE	0	NINE	Compare contents of Acc. 0 with '9' and branch if equal
BCS		START	
LDN	4	INPUT	Transfer input area characters to output area
LDN	5	OUTPUT	
MOVE	4	20	
PERI	0	CPCON	Peripheral instruction to punch a card
BRN		START	Unconditional Branch to read the next card

N.B. The flow-chart boxes and arrows would normally stand alone without the instruction lines and explanation which we've shown here.

I.C.T. 1900 SERIES

PLAN CODING SHEET

TITLE CARD REPRODUCING

PROGRAMMER A. BENNETT

LABEL	OPERATION	ACC.	OPERAND	
#PROGRAM			CARE	
#PERIPHERAL			CRO,CPO	
#LOWER				
			INPUT(20),OUTPUT(20)	
#LOWER				
CRCON			3/0,+0,+80,0/INPUT,0	
CPCON			4/0,+0,+80,0/OUTPUT,0	
BLANK			#20	
NINE			9	
#PROGRAM				
#ENTRY			0	
START	PERI	0	CRCON	[READ A CARD
	SUSBY		CRO	
	SUSBY		CPO	
	LDX	0	INPUT	[LOAD CC1 INTO X0
	SRL	0	18	
	TXE	0	BLANK	[TEST CC1 = SPACE
	BCS		STOP	
	TXE	0	NINE	[TEST CC1 = 9
	BCS		START	
	LDN	4	INPUT	[MOVE
	LDN	5	OUTPUT	[INPUT DATA
	MOVE	4	20	[TO OUTPUT AREA
	PERI	0	CPCON	[PUNCH A CARD
	BRN		START	
STOP	SUSWT		CR	[STOP TYPE 'HALTED CR'
#COMPLETE				
#END				
#FINISH				

Notice the Conditional Branch instructions, both BCS's, which cause the program to 'jump'. Notice, too, the Unconditional Branch, BRN START, after the card punching instruction.

These, and all the other actual instruction lines, give the program the logic and sequence which we devised by flow-charting.

Don't be put off by unfamiliar lines in the coded program. Directives, Data Statements and even unfamiliar instruction lines aren't important to our purpose—to write these lines you merely follow the rules for the specific computer's programming language.

The flow-chart shows the simple 'logic' of the job. That 'logic' is faithfully followed in writing the instruction lines.

So, if you **were** confused by unfamiliar lines in the example, go back, ignore them, and consider only the flow-chart and the way in which the instruction lines follow the pattern it lays down.

Let's think for a moment about the computer programmer himself.

Before you try your hand at the end-of-section test, let's try to summarise the qualities he needs:

1. He must, of course, have basic computer knowledge of the sort set out in this text.

2. He must have a special aptitude for analysing processes into logical stages.

3. He must be familiar with the common data processing jobs.

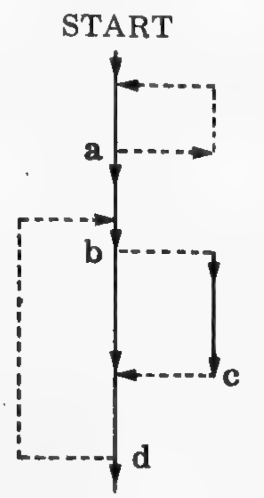
4. He must be thoroughly familiar with his specific computer's programming language. He must also have this knowledge in reference form—he needs to check the details of it, constantly—together with good access to the many odd technical points which will inevitably arise.

5. He must be able to take a broad imaginative view, although he's keenly concerned with technical detail and has the qualities which enable him to work to detailed completeness.

6. He must have an india-rubber!

CHECKPOINT

1. In the following flow diagram solid lines represent series of instructions. Write down the sort of instructions which must be written at points **a**, **b**, **c** and **d**.



2. A series of instructions must be performed eight times in succession.

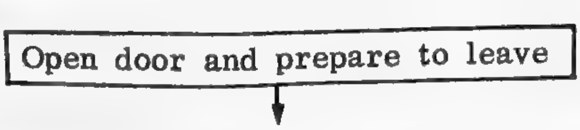
   (i) What is this bit of program called?

   (ii) What programming device would ensure that the correct number of passes are taken?

3. The program is obeying an instruction which is neither a branch nor a stop. Where in Core Storage must the next instruction be found if the program is to continue?

4.  What sort of operation is indicated by this flow-chart box?

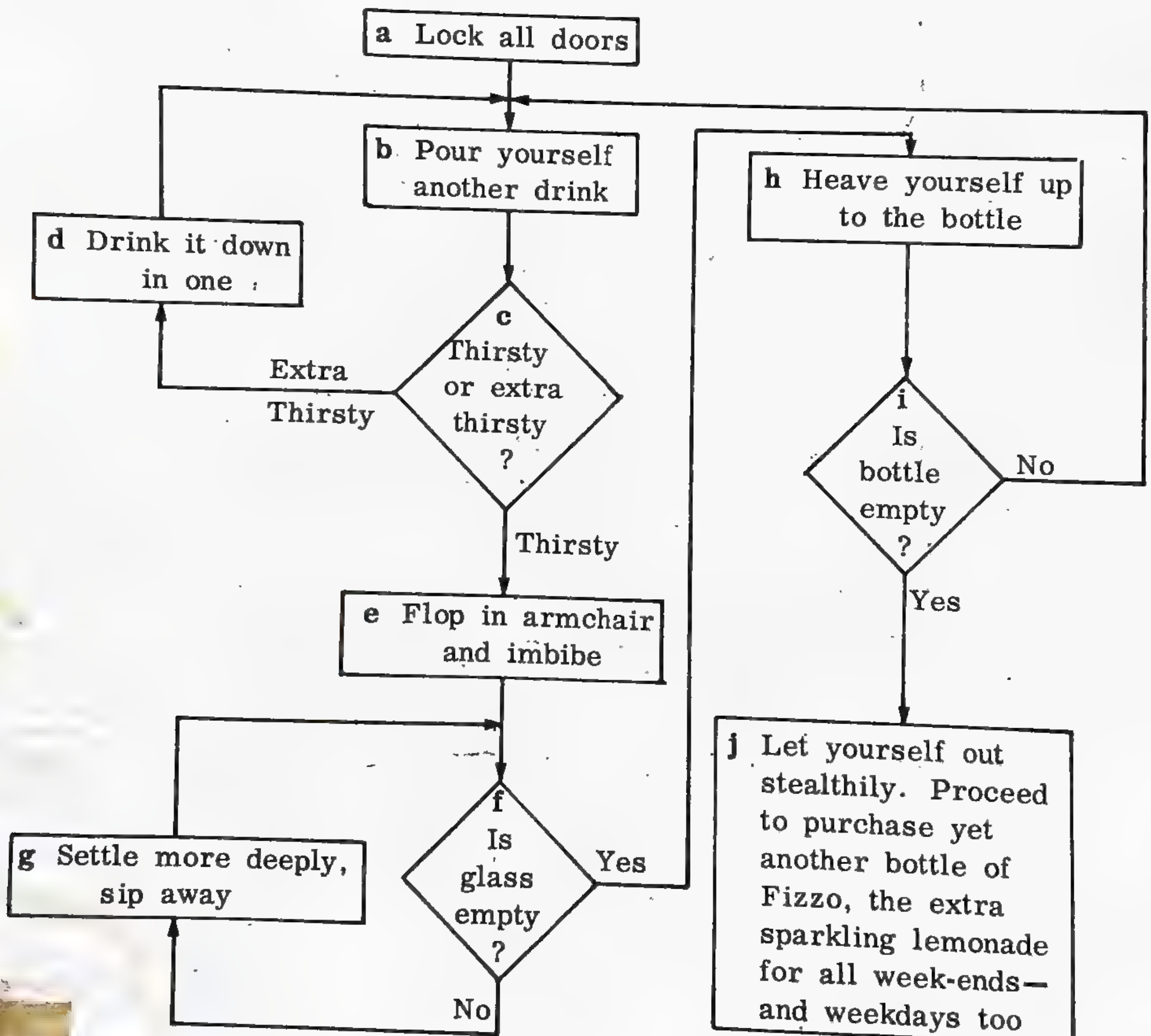
5. You always open the door to leave home in the morning without your raincoat or umbrella. If the weather looks fine you proceed to work. If the weather looks doubtful, you get your raincoat and then proceed. If it is actually raining, you get both your raincoat and umbrella before proceeding. Show this by means of a flow-chart, starting:



CHECKPOINT (Continued)

6. If the following were a **computer** program flow-chart, in what order would you write the instruction lines? Use the letters **a**, **b**, **c**, etc., from the boxes to represent the instruction lines, and insert UB where you'd need to write an Unconditional Branch instruction.

LOST WEEK-END



[Advertiser's Announcement]

CHECKPOINT (Continued)

7. On an actual computer program flow-chart, how many instruction lines are represented by each flow-chart box?

8. A source program is written in programming language.

   (i) What do we call the program which 'translates' the source program into machine terms?

   (ii) What do we call the resulting program which is loaded into Core. Storage to do the job?

9. Would you expect to find lines other than instruction lines on the source program Coding Sheet?

10. Consider this proposition:

    'Johnson is a bright lad, and very quick to analyse a job into its logical stages. He knows nothing about the XYZ Computer—or any other computer for that matter. But, I'll get him to draw the program flow-chart for the job, and your programmer will then be able to get straight on with writing the Coding Sheet lines.'

    Would you agree?

CHECKPOINT ANSWERS

For each fully correct answer, 1. to 10., give yourself two marks, and judge for yourself whether a partially correct answer is worth one mark or nothing.

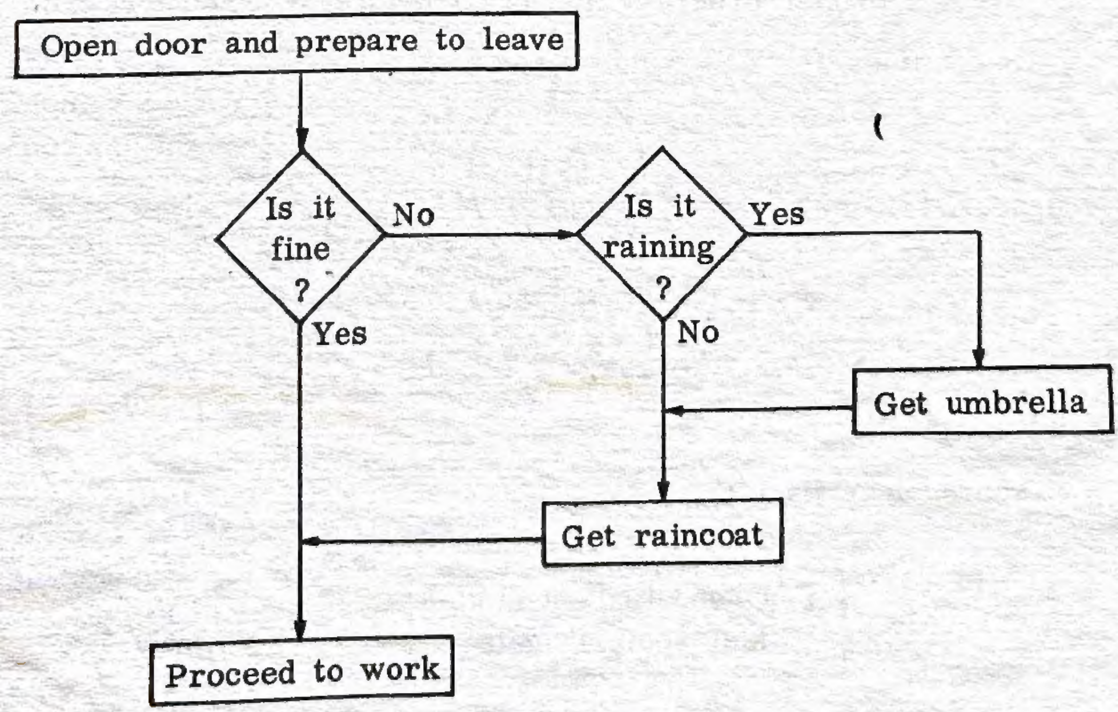
1. a Conditional Branch
   b Conditional Branch
   c Unconditional Branch
   d Conditional Branch.

2. (i) A LOOP (ii) A COUNT.

3. The next sequential Core Storage word.

4. The decision (or branch) box.

5.



CHECKPOINT ANSWERS (Continued)

6. (a, b, c, e, f, h, i, j) (d, UB) (g, UB). The brackets serve merely to show the main block of instructions and the two 'branch to' blocks. These blocks may be written in any order, but the sequence **within** the blocks must be kept. Regard more tortuous solutions as incorrect.

7. Any number (or 'One or more').

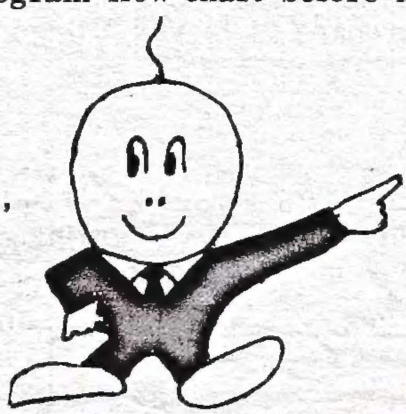
8. (i) ASSEMBLER PROGRAM.

   (ii) OBJECT PROGRAM.

9. Yes. (Because the program's various requirements must be stated by Directives and Data Statements.)

10. No, disagree. (It is most likely that Johnson's flow-chart would plan the job in computer terms. The programmer would need to make a proper program flow-chart before he could write the Coding Sheet lines.)

If you scored 15 or more,



You've completed the course

If you scored less than 15,

You need to look at this last section again

Other programmes available:

I.C.T. 1500 Machine Code Programming

1500 Assembly System (FAS)
Magnetic Tape Assembly (MTA)
File Control Processor (FCP)

I.C.T. Rapidwrite

Other programmes available on the ICT 1900 SERIES